Université Joseph Fourier — Master 2 Recherche – Systèmes et Logiciels

Dynamic instrumentation for application management and application analysis in component-based applications

Project by :

Walter RUDAMETKIN

Presented :

September 6, 2007

SARDES Project

INRIA Rhône Alpes – Laboratoire d'Informatique de Grenoble

Tutors :

Renaud LACHAIZE

Vivien QUÉMA

JURY :

Jean-Claude FERNANDEZ	Permanent member of the Jury
Jérôme EUZENAT	Permanent member of the Jury
Jean-Marc VINCENT	Permanent member of the Jury
Phillipe GUILLAUME	External examiner (STMicroelectronics)
Renaud LACHAIZE	Tutor
Vivien QUÉMA	Tutor

Abstract

Applications are becoming more and more complex. Tools that provide support for both, *application analysis* (e.g., profiling, workload analysis, software tracing) and *application management,* which are applications used to administer and control other applications (e.g., meta-applications, interposition tools, context propagation), are essential for modern applications. These two subjects, analysis and management, have traditionally been considered separate, but they rely on the same basis: *application instrumentation.* We unify application management and application analysis by providing fine-grained, dynamic instrumentation for component-based applications. Component models provide a well defined architecture, introspection and reconfiguration capabilities, that we utilize for instrumenting the application. We propose *Requests* as the base granularity for instrumentation. Moreover, we have proposed a novel *application management infrastructure* (i.e., meta-application infrastructure) that uses requests as its base granularity and provides two novel types of metadata, *request contexts* and *message contexts.* The infrastructure has been developed using the Fractal Component Model, and implemented in its reference implementation, Julia.

Keywords: application analysis, application management, meta-applications, context propagation, request, request tracking, profiling, workload analysis, component-based applications.

Résumé

Les applications modernes deviennent de plus en plus complexes. Des outils ont été développés pour les analyser (traçage de ressources, caractérisation de charge, ...) et les administrer (construction de méta-applications, propagation de contextes, ...). Ces outils ont jusqu'à présent été considérés indépendant et ne partagent ainsi pas de briques de base communes. Néanmoins, il apparaît que ces deux types d'outils nécessitent l'utilisation de techniques d'instrumentation. Dans le travail présenté dans ce rapport, nous avons étudié la possibilité d'unifier les outils d'analyse et d'administration d'applications. Pour ce faire, nous proposons d'utiliser comme base de ces outils une technique d'instrumentation dynamique à grain fin. Cette technique fonctionne sur les applications développées à l'aide de modèles de composants. L'intérêt d'utiliser de tels modèles est qu'ils fournissent des moyens pour introspecter et reconfigurer dynamiquement les applications, ce qui est une aide au développement de techniques d'instrumentation. Nous présentons également, dans ce rapport, une infrastructure d'administration reposant sur les techniques d'instrumentation précédemment citées. L'administration se fait à la granularité de la requête (de façon similaire à ce qui est fait dans les serveurs Web). Cette infrastructure a été développée à l'aide du modèle de composants Fractal.

Mots clés: analyse d'applications, administration d'applications, méta-applications, propagation de contexte, requêtes, traçage de requêtes, caractérisation de charge, applications à base de composants.

Table of Contents

Chapter I **- 1 -**
1 Introduction - 1 -
Chapter II **- 3 -**
2 State of the Art - 3 -
2.1 Overview - 3 -
2.2 Application Analysis - 4 -
2.2.1 Overview - 4 -
2.2.2 Software Tracing (DTrace) - 4 -
2.2.3 Dynamic Translation - 8 -
2.2.4 Workload Analysis - 8 -
2.2.5 Statistical workload analysis (Project 5) - 9 -
2.2.6 Deterministic workload analysis (Magpie) - 11 -
2.3 Application Management - 13 -
2.3.1 Overview - 13 -
2.3.2 Meta-applications and Metadata propagation - 13 -
2.3.3 Causeway - 14 -
2.3.4 Annotation Toolkits - 17 -
2.4 Summary of the State of the art - 19 -
2.4.1 Project summaries - 19 -
2.5 Remaining issues - 22 -
Chapter III **- 25 -**
3 Details of the Contribution - 25 -
3.1 Overview - 25 -
3.2 Synchronous interaction - 27 -
3.3 Asynchronous interaction - 29 -
3.3.1 Overview - 29 -
3.3.2 Defining annotations - 31 -
3.3.3 Proposed annotations - 32 -
3.4 Request tracking - 37 -
3.4.1 Overview - 37 -
3.4.2 Request execution paths - 38 -
3.4.3 Modifying request tracking granularity - 39 -
3.4.4 Request consumer mechanism - 39 -

3.5 Context propagation - 40 -
3.5.1 Overview - 40 -
3.5.2 Metadata key-value pairs - 40 -
3.5.3 Request context (global context) - 41 -
3.5.4 Message context (local context) - 42 -
3.5.5 Handling multiple contexts - 42 -
3.6 Callback infrastructure - 44 -
3.6.1 Overview - 44 -
3.6.2 Callback components - 45 -
3.6.3 Defining callback interaction points - 45 -
3.7 Profiling - 46 -
3.8 Comparison to other projects - 46 -
3.8.1 Comparison with DTrace - 46 -
3.8.2 Comparison with Project5 - 47 -
3.8.3 Comparison with Magpie - 47 -
3.8.4 Comparison with Causeway - 48 -
3.8.5 Comparison with Defensive Programming - 48 -
3.8.6 Comparison with A Posteriori Defensive Programming - 49 -
3.9 Summary - 49 -
Chapter IV - 51 -
4 Implementation - 51 -
4.1 Overview - 51 -
4.2 Implementation Context - 52 -
4.2.1 Fractal Component Model - 52 -
4.2.2 Julia, implementation of the Fractal Component Model - 54 -
4.2.3 FScript for Safe Dynamic Reconfigurations - 55 -
4.2.4 FScript Reconfigurations - 56 -
4.3 Application management infrastructure - 57 -
4.3.1 Trace Manager - 58 -
4.3.2 Request Tracker - 58 -
4.3.3 Callback Manager - 59 -
4.3.4 Meta-application Administrator - 60 -
4.3.5 Current state and remaining work - 61 -
Chapter V - 63 -
5 Conclusion - 63 -
Bibliography - 65 -

Figure Index

Figure 1: Abstract view of Magpie....................12
Figure 2: Causeway metadata concept....................15
Figure 3: Meta-application overview. 27
Figure 4: Dynamic tracers to instrument the application....................28
Figure 5: Hidden component functionality....................30
Figure 6: Annotation toolkit....................31
Figure 7: Thread pool/Thread creator....................34
Figure 8: Abstraction of message queues....................35
Figure 9: Multiple processes access data stream....................36
Figure 10: Request execution path....................38
Figure 11: Request context....................41
Figure 12: Message context....................42
Figure 13: Multiple contexts. 43
Figure 14: Conceptual view of the implementation....................52
Figure 15: External view of a Fractal component....................53
Figure 16: Internal view of a Fractal component....................53
Figure 17: Advantages of shared components....................54
Figure 18: Meta-application infrastructure....................57

Chapter I

1 Introduction

Modern applications are becoming more and more complicated and are commonly composed of hundreds of thousands of lines of code or are even in the millions of lines of code. This poses numerous challenges. Among them, we are interested in two, analyzing applications and managing them. *Application analysis* are the techniques used to study an application – its internal behavior, its external interactions, etc. It consists of a series of techniques like p*rofiling*, *performance debugging*, and *workload analysis.* A*pplication management* has different objectives. It attempts to separate the functionality of the application from how the internal software components interact and are used, by providing a means of controlling and administering the application. It rests on techniques like *context propagation*, *meta-applications* and *interposition.*

Application analysis and *application management* are currently considered separate. Consequently, existing solutions for application analysis and application management have created diversification in the way a developer understands the application and the way he actually interacts with it. More precisely, developers and application designers are forced to bridge the conceptual gaps between the different application analysis tools and application management tools because their relationships are not explicit nor clear. This comes from the fact that these tools do not share common base-concepts. This is particularly true for application instrumentation techniques, that are at the heart of both analysis and management tools. Instrumentation techniques used for application analysis are fine grain and do not only focus on inducing low-overhead, but also on becoming runtime dynamic. On the other hand, instrumentation techniques used for application management have focused on supporting *legacy* applications and are, consequently, coarse grain. Moreover, instrumentation is static, due to the fact that the application source code is not supposed to be available. Consequently, instrumentation for managing applications requires the modification of the context in which applications execute in order to intercept inter-application calls, operating system calls, and library calls. This large granularity of interaction between the application and the management tools have limited their use.

In the work presented in this report, we have studied the unification of the analysis and management functions. Our goal is to build an application management infrastructure that benefits from the techniques developed in application analysis environments. To that end, we, first of all propose a unification of the underlying application instrumentation technique. Instrumentation must provide a granularity that is useful to analysis and management tools alike. This means that application instrumentation must be fine-grained and provide interaction points in the application, where a developer needs and expects them. Unified instrumentation is the first step, but it is not enough, because the information obtained must provide an entity of abstraction that is not-only

beneficial to the tools, but must also be understandable by the developer, providing a means of quickly and easily relating all the inter-tool concepts, and a means of eliminating the conceptual gaps. We propose to use *Requests* as the granularity for interpreting application activity. A request is a single message sent to a software component for service. Requests can be split into smaller tasks and serviced by different software components simultaneously. One component may be servicing various tasks from different requests at the same time. These events must be analyzed so the request, as its own entity, can be constructed and used for analysis and management. We are not the first to propose requests. A *request-based* granularity for understanding workloads has already been proposed for web-servers. We extend the use of requests for analyzing activity in all applications, not only web-servers.

The the document is composed of five chapters. Chapter I Is the introduction we have just presented. Chapter II is a synthesis of the state of the art, concluding with a summary of the projects and the remaining issues that have not been addressed prior to this work. Chapter III details our contribution in the unification of application analysis and application management. Chapter IV shows an implementation of our infrastructure. Finally, Chapter V of this document shows our conclusions, including a summary of contributions and future work.

Chapter II

2 State of the Art

2.1 Overview

During the development of our project, we have studied areas of computer science that have commonly been considered distinct. Our solution provides a base for unifying these subjects. The subjects are: *application analysis* and *application management.* This chapter gives us an introduction to existing solutions in the domains of application analysis and application management. Special attention should be paid to understand the underlying instrumentation techniques used by all of these projects. Some are limited to passive analysis while others are disruptive to the applications behavior but provide a more thorough analysis. Application instrumentation is the basis for both analysis and management and provides the grounds for unifying them.

Instrumentation refers to the techniques to monitor or measure the level of performance, to diagnose errors and to write trace information from applications. Instrumentation is in the form of code instructions that monitor specific components in a system (for example, instructions that output logging information). Instrumentation is necessary to review the health and performance of the application. In general, instrumentation approaches can be of two types, source instrumentation and binary instrumentation. Instrumentation is the ability to incorporate tracing code, debugging code, exception handling code, performance counters and event logs into an application. Our interest is particularly focused on the ability to insert tracing code into applications and to provide interposition points. Tracing code serves the purpose of retrieving informative messages about the execution of an application at runtime. Interposition enables execution of external code at specific points in the application.

In section 2.2 we explain application analysis projects. We analyze the different approaches for instrumentation they use focusing on projects that have provided advances and novel techniques in the domain. We also take a view at application profiling, workload analysis and request tracking, which are techniques used for application analysis. In section 2.3 we explain application management. Application management refers to the main techniques used for interposition and meta-application construction. We explain the approach taken for instrumenting the applications and also how meta-applications are constructed.

2.2 Application Analysis

2.2.1 Overview

Application analysis refers to the techniques used to analyze and interpret an applications behavior. These techniques are varied, but all rely on application instrumentation. Among the techniques used, we will analyze projects in the domains of software tracing, dynamic translation, workload analysis, and application profiling. There are differences between the techniques, but many of the base concepts are the same, making some solutions appear to be a mix of different techniques.

In section 2.2.2 we will explain DTrace, a novel solution to software tracing. In section 2.2.3 we give the basic basic concepts and uses of dynamic translation in order to understand modern bytecode instrumentation techniques. In section 2.2.4 we will explain workload analysis and profiling. In this domain, we will analyze two different approaches, Project 5, which is statistical analysis using passive tracing that provides general information on application causality, and Magpie, which performs deterministic request tracking and provides performance modeling.

2.2.2 Software Tracing (DTrace)

Software tracing is a specialized use of logging to record information about a program's execution. This information is typically used by programmers for debugging purposes, and additionally, depending on the type and detail of information contained in a trace log, by experienced system administrators or technical support personnel to diagnose common problems with software. Tracing is a cross-cutting concern. Although there exist many projects on software tracing, the most notable to mention at the moment is Dynamic Instrumentation of Production Systems (DTrace)[Cantrill et al., 2004]. DTrace is a dynamic instrumentation system that unifies both user-level and kernel-level software in an absolutely safe fashion. We will introduce DTrace and provide a short reference to other projects in the software tracing domain.

Dynamic Instrumentation of Production Systems (DTrace)[Cantrill et al., 2004] emerges because performance analysis infrastructures have not kept pace with the shift to in-production performance analysis. Analysis infrastructures are still focused on the developer, on development systems, or both. They have rarely shifted to production systems. This causes problems because development systems differ from production ones, and it is a complicated task to replicate in-production systemic problems on development systems. In order to be a viable tracing infrastructure in production systems, the performance analysis infrastructure must have *zero probe effect* when disabled, and must be *absolutely safe* when enabled. Zero overhead when not enabled is key because production systems aim at maximizing resource utilization, especially since resources represent investment. Of course it is necessary also to minimize the overhead when enabled as to limit interference caused by the tracing infrastructure. Complete safety insures that executing the analysis infrastructure puts no danger to applications currently in execution.

DTrace has been integrated into the Solaris operating system and has been used to find serious systemic performance problems on production systems – problems that could not be found using preexisting facilities. In order to achieve its goal of dynamically instrumenting both user-level and kernel-level software in a unified and absolutely safe fashion, DTrace has developed a C-like high-level control language dubbed D for user friendly tracing. Some of the properties of the DTrace solution are:

- Dynamic instrumentation – when not in use it generates no overhead
- Unified instrumentation – user and kernel level software can be instrumented
- Arbitrary-context kernel instrumentation – virtually all of the kernel can be instrumented
- Data integrity – guarantees are provided on data integrity. Data is not lost nor altered.
- Arbitrary actions – the user can enable any probe with any action and safety is guaranteed
- Predicates – are used to record only necessary information
- A high-level control language – C-like language dubbed "D"
- A scalable mechanism for aggregating data – users may aggregate by virtually anything
- Speculative tracing – deferring the decision to commit or discard the data to a later time
- Heterogeneous instrumentation – Instrumentation providers are formally separated from the probe processing framework by a well-specified API, making it possible to use different instrumentation methodologies
- Scalable architecture – DTrace allows many tens of thousands of instrumentation points and provides primitives for subsets of probes to be efficiently selected and enabled
- Virtualized consumers – Multiple consumers can enable the same probe in different ways and there is no limit on the number of concurrent DTrace consumers

2.2.2.1 Detailed description of DTrace

The core of the DTrace infrastructure resides in the kernel. Processes become DTrace consumers by initiating communication with the in-kernel DTrace component via the DTrace library. The DTrace framework itself performs no instrumentation of the system; the task is delegated to instrumentation providers. For every point of instrumentation, providers call back into the DTrace framework to create a probe. Providers must specify the module name and function name of the instrumentation point, plus a semantic name for each probe. The probe identifier then consists of a tuple of 4 elements, `<provider, module, function, name>`, thus each probe is uniquely identified. Probe creation itself does not instrument the system, it simply identifies a potential for instrumentation to the DTrace framework. When a provider creates a probe, DTrace returns a probe identifier. Probes are then advertised to consumers, who can enable them by specifying any element of the 4-tuple. When a probe is enabled, an *enabling control block* (ECB) is created and associated with the probe. If there are no other ECBs associated with the probe (that is, if the probe is disabled), the DTrace framework calls the probe's provider to enable the probe. When a probe fires, that is, execution reaches the inserted and enabled probe, control is passed to the DTrace framework. DTrace makes no constraints as to the context of a firing. In order to assure safety from the DTrace framework, DTrace itself is non-blocking and makes no explicit or implicit calls into the kernel-under-study.

When the probe fires and control is transferred to the DTrace framework, interrupts are disabled on the current CPU, and DTrace performs the activities specified by each ECB on the probe's ECB chain. When all ECBs have been executed, interrupts are enabled and control returns to the provider. To simplify matters, all multiplexing of consumers on a single probe is handled by the framework's ECB abstraction. Each ECB may have an optional predicate associated with it. If an ECB has such a predicate and it is not satisfied, then processing advances to the next ECB. The

ECB is processed if the predicate is satisfied and it iterates over all of the actions defined in the ECB. Actions can indicate data recording, which is stored in the per-CPU buffer associated with the consumer that created the ECB. Actions may also update D variables. They are not allowed to store to kernel memory, modify registers, or make any arbitrary change to the system because that could risk destabilizing the system. Each DTrace consumer has a set of in-kernel per-CPU buffers allocated on its behalf and referred to by its consumer state. The consumer state is in turn referred to by each of the consumer's ECBs. When an ECB action indicates data to be recorded, it is recorded in the ECB consumer's per-CPU buffer. The amount of data provided by a given ECB is always constant, but different ECBs may record different amounts of data. Buffer free-space is verified before each recording so there are no buffer overflows.

Actions and predicates are specified in a virtual machine instruction set that is emulated in the kernel at probe firing time. The instruction set, "D Intermediate Format" or DIF is a small RISC instruction set designed for simple emulation and on-the-fly code generation. It is a design constraint that DIF emulation be absolutely safe since it is executed in the context of a probe fire. To assure basic sanity everything is verified and only forward branches are permitted. All loops are eliminated in order to disallow infinite loops. Many other safety features have been included to comply with the *absolutely safe* policy of DTrace. Some of them include prevention of certain memory loads. Run-time errors are handled. Hardware faults are handled. Even the kernel's page fault handler has been modified to recognize when a page fault has been generated from the DIF virtual machine.

2.2.2.2 Instrumentation Techniques

By formally separating instrumentation providers from the core framework, DTrace is able to accommodate heterogeneous instrumentation methodologies. Twelve different instrumentation providers have been implemented and none of them have any observable probe effect when disabled Having zero probe effect when disabled is a key feature of DTrace and makes DTrace dynamical in nature. Probes can be activated and deactivated when necessary. We will mention some of the providers included in DTrace:

- Function Boundary tracing – makes available a probe upon function entry and function exit. It is highly dependent on the architecture, requiring many modifications, including the C compiler. It has been implemented on SPARC and x86 systems using different techniques for each platform.
- Statically defined tracing – kernel code is modified and provide probe insertion points. You must be familiar with the kernel implementation to use it effectively. It is typically implemented by an expanding C-macro.
- Lock-tracing – this makes available probes that can be used to understand virtually any aspect of kernel synchronization behavior. It works by dynamically rewriting kernel functions that manipulate synchronization primitives, and is useful for understanding kernel resource contention.
- System call tracing – this provider makes available a probe at the entry and exit form each system call. It offers tremendous insight into application behavior with respect to the system. It works by dynamically rewriting the corresponding entry into the system call table when a probe is enabled.
- Profile – this provider is an unanchored probe that, instead of being associated with a point in execution, it is associated with an asynchronous event. The event source for this provider is a time-based interrupt of specified interval.

2.2.2.3 Interacting with the DTrace Framework

Using the D Language, DTrace permits users to specify arbitrary predicates and actions. D supports all ANSI C operations and allows access to the kernel's native types and global variables. D includes support for several kinds of user-defined variables, including global, clause-local, and thread-local variables and associated arrays. D programs are compiled into DIF by a compiler implemented in the DTrace library. The DIF is then bundled into an in-memory object file representation and sent to the kernel DTrace framework for validation and probe enabling.

DTrace provides user-level program instrumentation through the `pid` (process ID) provider, which can instrument arbitrary instructions in a specified process. The `pid` provider is slightly different from other providers in that it actually defines a class of providers. Each process can potentially have an associated `pid` provider. The techniques used by the `pid` provider are architecturally specific, but they all involve mechanisms to rewrite the instrumented instruction as to induce a trap into the operating system. The trap-based mechanism has a higher enabled probe effect than branch-based mechanisms used elsewhere, but it completely unifies kernel and user-level instrumentation. Any DTrace mechanism that may be used with kernel-level probes may also be used with user-level probes.

By tracing events in both the kernel and user processes, and combining data from both sources, DTrace provides the complete view of the system required to understand systemic problems that span the user/kernel boundary. Although DTrace is dynamic in the sense that it can insert probes at execution time and execute trace code, the probe emplacements are far from being dynamic themselves. Most applications require pre-runtime modifications in order to support different probes. This makes DTrace an efficient infrastructure but it is not completely free to insert instrumentation in all places. There are other interesting projects worth mentioning, although of less interest to us than DTrace.

2.2.2.4 Other application instrumentation projects similar to DTrace

Safety augmenting operating system execution with user-specified code has been explored in systems like VINO [Seltzer et al., 1996] and SPIN [Bershad et al., 1995]. More generally, the notion of augmenting execution with code has been explored in aspect-oriented programming systems like Aspectj [Kiczales et al., 2001]. However, these systems are all designed to extend the system but they were not designed for helping a user understand it. Systems like ATOM [Srivastava et al., 1994] and Purify [Hastings et al., 1992] instrument a binary and run it in place of the original. This allows a user to understand the system but their solution is completely static. Static solutions like these do not provide systemic insight, meaning they cannot integrate instrumentation from disjoint applications, and they are generally unable to instrument the operating system whatsoever. Also, instrumentation overhead remains constant during the full execution of the instrumented application. DProbes [Moore, 2001] is based on dynamic instrumentation and thus has zero probe effect when not enabled, but DProbes relies on a technique that is lossy when a probe is hit simultaneously on different CPUs, and misuse of DProbes can result in a system crash. Linux Trace Toolkit (LTT) [Yaghmour et al., 2000] is designed around a static methodology and introduces a small, but non-zero, probe effect at each instrumentation point. Arbitrary actions are not possible and the number of probes is limited to minimize generated overhead. K42 [Wisniewski et al., 2003] is a research kernel that has its own static instrumentation framework. K42 has lock-free, per-CPU buffering but K42 implements it in a way that sacrifices the integrity of traced data. Many of its limitations are the same as in LTT. Kerninst [Tamches et al., 1999] is a dynamic instrumentation framework of use on operating system kernels. It achieves zero probe effect when disabled, and

allows instrumentation of virtually any text in the kernel. However, users can accidentally instrument routines that are not actually safe to instrument and cause fatal errors.

2.2.3 Dynamic Translation

Dynamic translation, also known as Just-In-Time compilation (JIT), is a technique for improving the runtime performance of a computer program. It converts, at runtime, code from one format into another (e.g., bytecode into native machine code). The performance improvement originates from caching the results of translating blocks of code, and not simply evaluating each line or operand separately, or compiling the code at development time. JIT builds upon two earlier ideas in run-time environments: bytecode compilation and dynamic compilation. Several modern runtime environments, such as Microsoft's .NET Framework and most implementations of Java, rely on JIT compilation for high-speed code execution.

JIT compilers modify code that is to be executed at runtime. There are many purposes for doing this (e.g., code optimization, code portability) but we are interested in the ability to insert instrumentation directly into bytecode in order to better understand an application. One JIT compiler [Olszewski et al., 2007] instruments operating system code by overwriting the system function table entry corresponding to the function it wishes to instrument. The overwritten addresses then point to the JIT, which copies the code of the function to a new location and inserts instrumentation in a quick and dynamic manner directly into the code. All calls to the original function then pass through the instrumented version of the function, which is controlled by the JIT. When instrumentation is to be removed, the original system function table is rewritten with the original address of the non-instrumented function, and the instrumented version of the function is discarded.

Conceptually debuggers and JIT compilers are similar. Debuggers [GDB][Eclipse] are also a means for application instrumentation. When an application is compiled and the debugger instructions are enabled, trap and other instructions are inserted into the application into specific points. These instructions are then used to return execution control to the debugger. The debugger can choose the granularity of instrumentation by recompiling the application. The basic fault is that these instructions produce large overhead and are not dynamic. This is why an application should not be normally executed with debugger instructions enabled. The difference with JIT compilers is that instrumentation is performed at runtime and instructions are not only for halting execution of the application under study, but can be arbitrary instructions for many different purposes.

2.2.4 Workload Analysis

Knowledge of how a system processes a workload is very important in modern day systems in order to locate performance bottlenecks and problems. Complex systems are composed of many different software components that interact in ever more complex manners. These systems must be analyzed and tools provided in order to improve how the system is understood.

A common scenario is a distributed system that is composed of multi-vendor software components. A solutions vendor can provide a system built from different modules, but he does not necessarily have the expertise to correct all performance issues. These multi-developer solutions can have many performance problems that are hard and expensive to locate. Individual vendors may provide support and training for solving performance issues within their components, but not necessarily for solving cross-component issues. Many problems cannot be solved by only focusing

on single components, and require a system wide view. The tools provided should limit their requirement of direct support from the components they study, because many components are not provided with source code. This limits the solution to the borders of components and the interaction that each component has with the system. Also, tools should be automated as much as possible, requiring minimal human aid. The purpose of the tools is, in general, not to solve problems, but to isolate them efficiently and accurately, increasing programmers efficiency.

There have been many approaches providing insight on how to tackle this problem, but most are not dynamic or require severe modifications of the application under study. One approach [Hrishchuk et al., 1995] is to obtain causal traces of distributed systems and resource demands, by labeling each end-to-end activity using an object oriented prototyping language. This approach is interesting but not useful outside the prototyping system. An old version of Magpie [Barham et al., 2003] associates traced messages of incoming requests with a unique identifier, and associates resource usage throughout the system with that identifier. This requires a very sophisticated tracing infrastructure, but simple post-processing analysis. The Distributed Programs Monitor project [Miller, 1984] reports causality using kernel implementation to track the causal information between pairs of messages, rather than inferring causality from timestamps. A different approach is one that Netlogger [Gunter, 2005] takes, which is to require programmers to add event logging to carefully chosen points in an application, and then generate "lifelines" that respond to causal paths. Netlogger provides tools to visualize logs, but the tools are somewhat lacking.

We find two different groups of work interesting and close to ours. These groups are workload analysis based on statistics obtained from the application under study, and deterministic workload analysis, which follows requests through a system as they are being serviced. In the following sections we will give an introduction to both, statistical and deterministic, workload modeling, and describe the work done by two major projects in these areas. These projects are Project5 [Aguilera et al.,2003] which is funded by HP, and the newest version of Magpie [Barham et al., 2003], funded by Microsoft.

2.2.5 Statistical workload analysis (Project 5)

Project 5 [Aguilera et al.,2003] infer causal paths from message traces to locate nodes of a black-box distributed system causing performance bottlenecks. They provide tools for aiding in debugging performance bottlenecks in distributed systems of black boxes. Their system is based on Pinpoint [Chen et al., 2002], and estimates causality between nodes using statistical algorithms. They use two different algorithms, one for RPC style communication and another for message based communication, to locate the most likely parent node of an inter-node call. This is similar to constructing a call-graph, with the subtle difference that one node can be servicing various requests originating from the same node simultaneously. Since nodes are black boxes and no extra instrumentation is added it becomes difficult to know exactly which call is related to which return.

For example, in a system with only two nodes, A and B, suppose A calls B three times in a row before B has had time to finish any of the calls. That means that B is currently servicing 3 calls from A. B then returns 3 results to A. It is not clear which result from B belongs to which call from A because nodes are black boxes and the calls are performed across a network. If the system is implemented using RPC style communication, we would know that A has performed three calls towards node B. Then B has returned three answers to A. To complicate even more the matter, in a message based system we would only see three messages going to B and three messages going to A. In this context, it would not be clear if a message is an originating call or a return. This is because messages are sent to a node but there is no specific information that tells if the message is a call and expects a result, if it is a result, or if it is just information to be shared between nodes.

2.2.5.1 Approach

The approach taken by Project5 to infer dominant causal paths in distributed systems relies on tracing messages between nodes, and using offline algorithms to infer causality from these traces. The algorithms developed infer multi-hop causal path patterns, and provide statistics about latency. The message traces created are relatively simple, and the results are inferred statistically from the offline algorithms. As previously mentioned, two kinds of communication in distributed systems are taken into consideration. A causality analysis algorithm has been developed for each type of communication. The first one, the *nesting algorithm* for RPC-style (remote procedure call) communication, operates on individual messages in the trace. The other, the *convolution algorithm* for message-based communication, uses signal-processing techniques to extract causal information from traces. The task is difficult because a real trace contains interleaved messages from separate causal paths. The approach involves three phases:

I. Exposing and tracing communication. This phase happens online and is where a complete trace of all inter-node messages are gathered. These traces can be gathered under a real or synthetic load. Each trace contains, at a minimum, a tuple or series of tuples (time stamp, sender, receiver).

II. Inferring causal paths and patterns. This phase happens offline and is where one of the two post-processing algorithms are used. The algorithms must handle noise and incomplete traces, given the nature of a distributed system. An important factor to consider is that these algorithms need not be fool-proof, as they are only intended to help humans understand and debug systems, not to automate control.

III. Visualization. An important factor is being able to understand and visualize the information that has been analyzed. This phase has only been partially addressed so far and requires more attention.

The details of the two algorithms proposed are complicated and not necessary for the understanding of the rest of our project. Instead, we will provide a short comparison of the two algorithms.

2.2.5.2 Nesting algorithm VS. Convolution algorithm

Comparing the two algorithms comes down to comparing RPC vs free-form messages. The convolution algorithm can find causal relationships in any form of message-based system, while the nesting algorithm is only useful in RPC systems. Even further, the nesting algorithm, as implemented, is not useful in RPC systems with call forwarding or asymmetric returns, forcing the use of the convolution algorithm in these cases. The nesting algorithm provides a more concise representation of the system than the convolution algorithm. Also, rare events may be found using the nesting algorithm, which is not possible with convolution since causality is caught by looking at spikes in the correlation of two signals, and rare events do not create spikes. The traces gathered for the convolution algorithm are simpler, only requiring timestamps, sender id and receiver id. For the nesting algorithm it is also necessary to mark entries with RPC call or return. In some cases this extra information may be inferred. The algorithm also performs much better if the trace system can extract call identifiers from the RPC messages. Practical running times of the nesting algorithm are quite low (much lower than the duration of the traces themselves) and the space overhead is likely the limiting factor. The convolution algorithm has space complexity linear in the length of the trace. Running time is the dominant cost for the algorithm and can be much slower than the nesting algorithm. In practice, there is a trade off between precision of the delay results and longer running time.

2.2.5.3 Obtaining traces

In order for the algorithms to work, it is necessary to obtain traces from the system. There are numerous challenges when attempting to trace all messages between nodes in a distributed system. Black-box assumptions simplify the tracing problem, because relatively little information is needed about each message. The black-box solution proposed by project 5 implies that absolutely no support be required from the nodes of the system, no specific message knowledge be known, and it does not perturb system performance at all. This is because they do not suppose software vendors agreeing on any particular solution, so the applications-under-study must not be modified.

Passive network tracing can approximate this ideal, but cannot always expose nodes at the appropriate level of granularity. Two techniques have been used for achieving passive tracing, namely Port Mirroring, and Packet Sniffing at each participant host. In the cases where passive network tracing does not obtain the amount or the detail of information required a more intrusive tracing mechanism must be implemented. It is true that this compromises some of the black-box ideals, but if the costs remain low enough the tools may still be useful. It is even possible to have passive networking traces and more intrusive traces merged, building a unified view of a complex system built from new and legacy components. This allows programmers to create traces in specific areas of interest inside their components and still understand the overall interactions of the system.

Overall, project5 adheres to a black-box ideal very well. They provide causality analysis in distributed systems in a novel and dynamic way. Their dynamicity comes from being able to start tracing and remove tracing at any moment of execution and, in the case of passive tracing, this has zero overhead.

2.2.6 Deterministic workload analysis (Magpie)

Magpie [Barham et al., 2003] attempt to correlate events in a system and relate them to the treatment of a request, providing an interesting method of analyzing overall system interaction and also the capability of focusing on individual requests in order to distinguish individual work elements, like those that cause bizarre or aberrant behavior. A request is defined as system activity which takes place in response to an action initiated by the application being traced (e.g., HTTP request, database query, file open request). A request is described as the sequence of applications involved in its processing and the resource consumed at each stage (e.g., CPU, bandwidth, disk transfer, latency). Request tracking is inherently difficult because of various reasons. There are many software components involved that are spread across different machines. No globally unique request ID is present. Multiple thread pools are used to service requests. Threads interchange request information asynchronously and many synchronization primitives occur in user-mode and are not visible by the OS or OS libraries. Magpie focuses on distributed systems or more precisely, multi-tier systems, where elements of the request will be serviced on separate machines and separate applications. Their work has been implemented on a typical web-service, where part of the request is serviced on one machine and another part is serviced on a distant machine. A basic example of this is a web-server running on a frontend machine and a database server running on a backend machine.

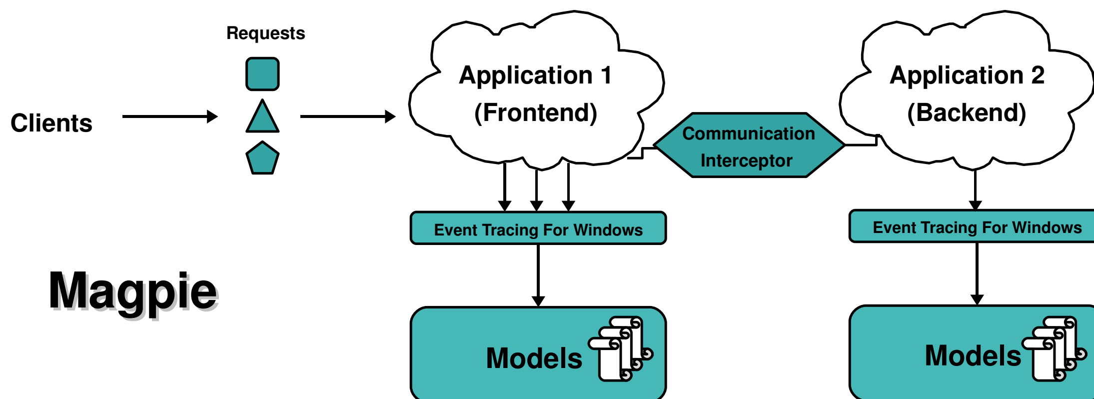

Figure 1: Abstract view of Magpie

Shows a high level and simplified view of Magpie. The system is composed of two applications, a Frontend and a Backend. Requests are received at the frontend and the request is serviced by different threads simultaneously. All communication channels are intercepted. Models are specified by an expert of the system and used to interpret and relate the events from ETW (Event Tracing for Windows) to the request in question.

Magpie logs events belonging to a particular request and performs *temporal joins* over the log of events in order to identify the requests and to analyze the resources consumed by that request. Magpie relies heavily on an event infrastructure already integrated into the Windows operating system. They utilize an application specific schema to correlate events obtained from the Windows Event Tracing infrastructure. This application schema relates events in the system produced from instrumentation, interception and EWT (Event Tracing for Windows), but no global request ID is required. The application schema is not automatically constructed and a certain degree of expertise is required in order for it to be constructed. Part of the problem relies on the fact that a single request is serviced simultaneously by different software components. This occurs because requests are divided into smaller elements and execution is parallelized. It is necessary to understand how the software components interact and what each event produced from the system really means. For example, which function calls cause a thread from a thread pool to be activated or send a message to a message queue for later servicing. The application specific schema then correlates activities in the system using thread ID, function name, function parameters, time of the event, and other elements in order to relate the activity to a particular request. Errors in the schema cause incorrect attribution of requests and resources. Also, this becomes exceedingly difficult because Magpie works on legacy applications of which one does not usually have access to the source code, such as IIS (Microsoft's Internet Information Server). This difficulty in itself, which is to completely understand an application of which you do not have access to the source code, makes Magpie's solution unfeasible except for people with advanced expertise in the system and only in limited cases.

Part of the Magpie project is relating events in the system to identify a request. This is done by observing the control flow (causality) in the application and by analyzing resource consumption. This provides a fine-grain way of debugging performance issues. The other part of Magpie's work is based on workload analysis and profiling of the application in order to analyze applications in a more coarse-grain manner. This is done by clustering similar requests based on their behavior and by building a probabilistic workload of the aggregated requests. This information is later used for performance debugging (e.g., fault detection, configuration, management), and for performance prediction (e.g., realistic workload models, capacity planning).

2.3 Application Management

2.3.1 Overview

Application management attempts to extract the administration of an application from the functional concerns. It is beneficial in creating generic software components that are reusable, because the components can focus on functionality (e.g. their specific job), and not on non-functional concerns (how they interact e.g., security issues, priority, QoS). There are a couple of techniques used for achieving application management. Among them, there is *interposition* and *meta-modeling*. Interposition focuses on interrupting execution of the application at a specific point, and executing predefined functionality. Interposition can be seen as a rudimentary and non-generic way of constructing meta-applications. Meta-modeling is complicated to define in few words because there is not a true consensus to what it is. For our purposes, let us say that meta-models are implemented by means of *meta-applications*. Meta-applications are the entities that will control the applications themselves and perform decisions related to application execution. Meta-applications require information related to the data that is being serviced in order to achieve their tasks. This information is called *metadata.*

In section 2.3.2 we explain meta-applications and metadata propagation in general, and we give a thorough analysis of Causeway, a meta-application infrastructure. In section 2.3.4 we explain annotation toolkits. They are a fine-grained, inter-code way of providing basic interposition and control execution flows of an application, but they are not generic.

2.3.2 Meta-applications and Metadata propagation

Metadata is information that is associated with data that is currently being serviced. One definition of metadata is *"Metadata is structured, encoded data that describe characteristics of information-bearing entities to aid in the identification, discovery, assessment, and management of the described entities"*. Loosely, what this means is that data that exists in the application and is being serviced can have other data that explains it, commonly understood as *higher-level* data. The difference between metadata and the data that is in service is that metadata is non-functional data in the application, that is, it is information that is not required by the component in order to complete its particular task. Metadata is normally grouped into a context and is useful for evaluating non-functional concerns in an application. These concerns can include, but are not limited to, security concerns, QoS, application performance, and many others. Keeping this sort of information outside of the application keeps the application itself clean with no extraneous API usage, and also allows the addition of information to read-only components, such as 3rd party components.

Context propagation allows programmers to associate information with functional data. The metadata is hidden from the functional elements of the application and is used when non-functional decisions are to be made. These decisions are performed by meta-applications. These meta-applications should only interact with the non-functional data of the application, since they are to be used for non-functional concerns of the application. Traditionally, there have been two approaches to writing meta-applications: a log-based approach, and a metadata based approach. Log-based approaches generally operate in two phases – first, execution events are recorded in logs, and next, the log records are analyzed. The analysis of logs can be performed while the application is executing or they can be performed postmortem. A limit to a log based approach is that the execution of requests cannot be altered because processing of the events lags the execution.

Metadata propagation is done inline with thread execution so meta-applications can access this information and control or alter the applications execution. These execution modifications are done to satisfy the non-functional concerns of the application previously mentioned. The authors of Causeway adopt the metadata passing approach, hence their objective is to provide a framework that makes meta-application development easier.

There are two projects of interest that have worked on automatic context propagation and meta-application construction. These projects, namely SDI [Reumann et al., 2004] and Causeway [Chanda et. al, 2005], are very similar to one and other. Causeway is a project that implements almost all the same concepts as SDI and adds a couple of improvements. In order to avoid repeating two very similar projects we will only explain Causeway.

2.3.3 Causeway

Causeway is an attempt to build a general meta-application infrastructure for multi-tier applications. These applications are more common than ever and are composed of multiple programs communicating among themselves that can be spread across multiple machines. Requests are then serviced by multiple execution threads running in different software components. The approach is to propagate metadata with request data so that meta-applications can use the metadata to achieve various goals. These goals should be restricted to non-functional concerns only.

Causeway provides an interface to associate metadata with threads and facilitates the propagation of such metadata across communication channels. Causeway manages, handles and propagates metadata transparently so meta-applications can be easily built on top of it. Applications that do not require metadata remain oblivious to the contexts being passed, even though the context exists and there is a constant overhead generated from its treatment. An alternative to Causeway is to augment all application-level interprocess communication protocols as to transport metadata. This implies modifying every function call as to make metadata directly visible and propagated by the applications. Causeway chooses to make metadata propagation an operating system-level function, to make it independent of the application-level communication protocol used. This gives Causeway a large advantage, since not all applications in a multi-tier system have to be metadata-aware.

Causeway proposes automatic propagation of metadata across *system-visible* communication channels. By system-visible, we refer to those implemented directly in the system kernel and system libraries (e.g. sockets, pipes). For *non-system-visible* channels (e.g. shared memory) Causeway provides an API to be called from application code in order to transfer metadata.

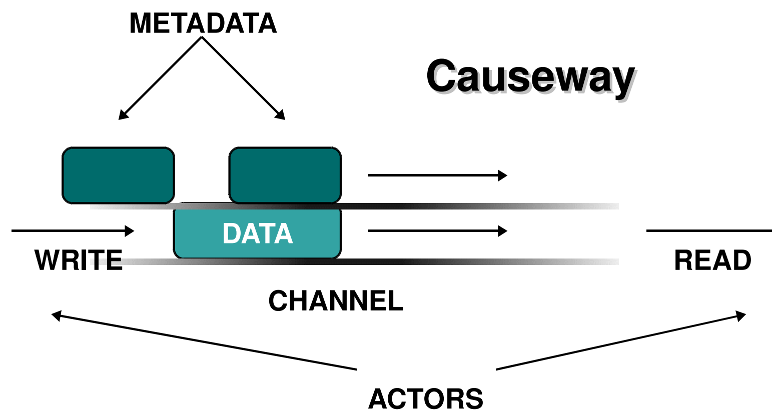

Figure 2: Causeway metadata concept.
Shows Causeway's basic concept of metadata propagation. Metadata is directly associated with functional data, and at every interprocess communication (IPC) point metadata associated to the writer is copied to the channel. Readers of the channel receive the functional data and are associated to the metadata. Metadata propagation is handled automatically by the Causeway framework.

Causeway uses an interface for injecting, inspecting, modifying and removing metadata. Metadata is originally assigned to a thread. When a thread sends request data to another thread along a channel, Causeway transfers metadata from the former thread to the latter. Support for metadata propagation is required at *transfer-points* where an application thread sends or receives data from a channel.

Causeway consists of two parts in total:

1 Interfaces used by applications to manage and utilize metadata

2 Mechanisms for propagating metadata

2.3.3.1 Managing Metadata

Metadata in Causeway is a tuple containing the *metadata-type* and *metadata-value*. Metadata types include request priority, request identifier, and security identifier. Other metadata types can be directly specified by the meta-applications. Metadata is managed in a dictionary keyed by the address of the associated *entity*. An entity can either be a thread of control or data that is read from or written to a channel. A thread's metadata is propagated with the data written to a channel on a write operation. When the data is read, the associated metadata is then propagated to the thread performing the read.

Metadata can be assigned to a thread in two manners:

1 Metadata injection, using the Causeway API

2 Metadata propagation, when reading data from a channel

It is important to know that newly assigned metadata replaces the thread's metadata of the same type. This implies that metadata can come from one destination, as to say that a thread is treating one, and only one, request at any given moment.

2.3.3.2 Meta-application interaction points

Meta-applications have two distinct ways to interact with Causeway. The first is through an interface to inject and modify metadata, and the second is through a callback interface, in which Causeway passes execution to handlers that have been properly registered by the meta-application. The metadata interface can be called from user or kernel level threads. It consists of four functions that enable metadata to be inspected and/or modified. The functions are: `cw_type_query`, `cw_data_lookup`, `cw_data_insert`, and `cw_data_remove`. The function names are self explanatory so we will not go into more details. These callbacks are points in the kernel or kernel libraries where metadata is passed between execution threads. A meta-application can register a callback method to a certain *transfer-point*. Transfer points are points where data is read or written to a channel by a thread. When metadata passes through a transfer point the registered callback methods are invoked with the metadata as an argument. The order in which the callback methods are executed is important because a callback method may modify metadata or alter execution. A registered callback at a transfer point is executed and has access to the metadata and to execution decisions. For example, if a security ID is lower than necessary, execution of the request can be immediately terminated.

2.3.3.3 Propagating metadata

Metadata is originally associated with a thread. When a thread performs a transfer of data across a communication channel, its metadata is associated with the data that is sent. The thread that receives the data is then associated with the corresponding metadata from the data he received. In essence, this should happen for all inter-thread and interprocess communication. Metadata transfer is done using a System Programming Interface (SPI) that consists of a single function. The function provided is `cw_metadata_xfer`. It obtains the source entity's metadata and transfers it to the destination entity. At any transfer point a single call to `cw_metadata_xfer` suffices for metadata propagation.

The transfer points that have been instrumented in the implementation of Causeway are the following:

1. User-level to kernel-level thread
2. Kernel-level to user-level thread
3. Kernel-level thread to message
4. Message to kernel-level thread

Causeway handles sockets and pipes similarly. When a thread writes to a socket or a pipe, the thread's metadata is associated to the data written via the metadata transfer SPI. This is true only for local sockets. When using Internet sockets, the data is encapsulated in IP packets for send and receive across sockets. Causeway encapsulates metadata, in addition to data, in the IP packets.

System-opaque channels occur inside of the application. They are shared-memory channels that are not visible to the augmented kernel nor to the augmented kernel libraries. For metadata propagation to be consistent for the whole multi-tier application, these channels must also be instrumented. It is very difficult to automatically instrument shared memory channels because they are not easy to locate, and even if a shared memory location is found, there are instances when it is incoherent to propagate metadata because the shared memory does not imply data transfer (e.g. application specific memory allocator). For the above conditions this is not done automatically in Causeway, but Causeway alludes to locating synchronization mechanisms and analyzing them as a

possible solution to automating instrumentation of shared memory channels. In any case, the application must be instrumented using the `cw_metadata_xfer` function previously mentioned.

2.3.4 Annotation Toolkits

Annotation toolkits are aids for programmers in order to express functionality different then that of the code itself. This can be to add commentaries for creating automatic documentation, to add compiler directives for optimizations, to add virtual machine directives that can be interpreted at runtime, or any number of different activities.

We are interested in annotations that are used for a specific purpose, meta-functionality. In particular, the annotations that interest us are those that automate non-functional concerns of the application. We show two different projects that use annotations to express DoS resistance. The first project inserts the annotations directly into code, and the second one, which is limited to component-based systems, adds annotations to external files which construct the component application. Although we are not directly interested in security or DoS attacks, these annotations give us a manner to make visible application activity that would normally remain obscure. They also show insight on the feasibility and possible design of an annotation toolkit for our purposes.

2.3.4.1 Denial of Service (DoS)

Denial of Service (DoS) attacks are a major source of concern in the Internet. DoS attacks are designed to consume a disproportionate amount of resources on the target system by exploiting weakness in the network software. Such attacks can cause the systems performance to slow down and even make the system unavailable for well behaving users. Protecting code from DoS attacks is often considered the responsibility of the OS, firewalls and intrusion detection systems. As a result, many DoS vulnerabilities are not discovered until the system has been attacked and the damage is already done. Defensive programming [Qie, 2002] proposes and describes a software toolkit to improve robustness of code against DoS attacks. It is important to find automated ways of protecting software. Many factors are involved in making software DoS resistant and, to make things worse, most DoS attacks are unknown at the time of development. Creating robust and DoS resistant software is a challenging task. We will explain the solutions that rely on writing ad-hoc functions to constantly monitor the applications execution flow. Two approaches are studied. One that annotates the source code with macros that control how a function is used, and may deflect the execution path in case of suspected abuse. The other annotates external application-design files for the same purposes.

2.3.4.2 Inter-code annotations

Inter-code annotations are used by an annotation toolkit [Qie, 2002], which attempts to provide an automated defense against DoS attacks. The proposition is called defensive programming, by which it is suggested that a programmer must take an active role and provide systematic proactive protection against DoS attacks by embedding general mechanisms into software. Ideally, defensive software can protect against previously unknown DoS attacks. The key idea is to insert annotations that monitor and control the execution of the program at runtime. These annotations serve both as *sensors* that detect anomalies and *actuators* that change the control flow of a program when they detect that defensive maneuvers are necessary. For this purpose, a toolkit has been developed consisting of a set of annotation primitives, a runtime library, and a set of compiler extensions.

Programmers can now specify where resources are acquired/consumed/released, where the program branches into independent functionalities, and what principals are holding resources. Rather than focus on implementation details, programmers are asked to identify the services provided and the resources consumed by their program at a high level. The effectiveness of the toolkit depends on whether a good defense policy can be specified, which is ultimately the responsibility of the programmer. However, significant improvement of software robustness can be achieved with relatively low programming effort. To implement the defensive strategy, experienced programmers annotate source code of the application with macros. Then, during the programs execution, macros control how a function is used and, in case of abuse, deflect the execution path to some other function. Placing the annotations in the right point is a sensible task, on which all the rest of the system relies. This is also the only possibility of spotting out potential weaknesses in the program. In case of misplacement of the annotations, the system will be susceptible to attacks. The toolkit itself proves the feasibility of annotations, but the particularities of the annotations themselves are not required to be understood because they are DoS specific.

The toolkit is implemented using a different C-macro for each annotation and is linked with an instance of a corresponding data structure that represents the annotation. There are situations where the toolkit has difficulty in providing protection to the desirable level. Some of these situations are implementation issues that can be improved while others are fundamental limitations of the approach. The current toolkit only applies to a single process because the sensors and actuators need to share state, and thus, they only work within a single memory space. It is not sure how inter-process communication may help or hinder. Finally, rate limiting, which is the tactic that the annotations employ, only controls the quantity of resources consumed by each service, but not the order that resources are consumed. Not being able to schedule resources may lead to more conservative specifications in resource limits, and should be improved upon.

2.3.4.3 External annotations

The previous solution relies on annotating the source code directly via macros. Placing these macros or annotations is a sensible task. A-posteriori Defensive Programming [Schiavoni, 2006] proposes a different approach, but it is restricted to component-based systems. They show that in this context, a general mechanism to detect DoS attacks is possible. The key idea is to annotate services and use the annotations to detect an attack. In component based systems, a component exposes its services, so in essence, the components themselves must be annotated. These annotations can be applied *after* the design and implementation of a component and without modifying either of them. It is even possible to add DoS protection to an already deployed application. This gives way to the *a posteriori* naming of the approach.

Component-based systems are made of an assembly of components that interact through *bindings*. Bindings are established between components requesting a service and components providing a service. It is easy to look at bindings to find components that provide resources and components that consume resources, helping to identify components that need to be defended and making component isolation much easier (isolation being an important defense strategy). The proposal is to use annotations at the design level. Annotations are metadata that mark component interfaces and allow expressing semantics about a given component. An *Overlay of components* is a set of components that are marked by the same annotation. All the services belonging to the overlay are protected by the same defense strategy. A defensive strategy is defined by an *annotation consumer* whose role is to detect which semantics have to be applied to the component annotated with the annotation it is in charge of. When the annotation consumer is deployed with the rest of the application, it will monitor the activity of all the components in the same overlay. Once the overlay is defined, the annotation consumer implementing the policy to deploy is completely

independent of it. The application deployer only has to annotate each service. Being that required and provided services are explicitly declared this effort is greatly simplified.

The annotation toolkit has been implemented within Fractal. Fractal is a Java-based component model that provides an Architecture Description Language (ADL), which allows the description of component configurations. The fractal ADL has been extended adding an *annotatedby* attribute to the *interface* element. This attribute is used to define which annotation is to be used for a given interface. Furthermore, the implementation relies on Aspect Oriented Programming (AOP) techniques and on Java 1.5 annotations. Java 1.5 annotations are first class elements, themselves annotatable by meta-annotations. Through meta-annotations, two parameters are specifiable: *target*, to specify the granularity of an annotation and, *retention*, indicating how long an annotation has to be retained. The ADL is parsed by a factory that produces components whose interfaces are annotated using annotations specified in the ADL description. Annotation consumers, on the other hand, are developed using AOP techniques. I will give a very brief explanation on AOP (Aspect Oriented Programming). AOP implements crosscutting concerns that affect several classes and that are not well modularized. It allows the implementation of these concerns in well modular well-localized entities called *aspects*. Aspects are made of dedicated constructs that mirror well-defined points in program flow and structure. This is called the *joinpoint* model. A *pointcut* construct lets you pick out join points that match a certain criteria, and an *advice* construct lets you add code to be executed at those points. To better understand the concepts, there is an implicit relation between *sensors/activators* and *pointcuts/advices*. The aspect using an annotation to define an interesting point in the execution flow of a program is called *annotation consumer*.

The final element to be solved is deployment of aspects and components. The Fractal component model provides a runtime environment that allows creating components from their ADL definition. Unfortunately, because part of the code of Fractal components is dynamically generated, it is not possible to intertwine aspects and components source code. The chosen solution was to modify Fractals runtime (not extend Fractal) to make use of load-time weaving mechanisms introduced in AspectJ5.0. By focusing on component based systems it is possible to provide a general mechanism to detect DoS attacks. The provided solution gives two important advantages over other proposed solutions: (1) source code does not require modification and, (2) it can be applied at deployment time.

2.4 Summary of the State of the art

In this section we will provide a short summary of the projects explained in the state of the art, mainly focusing on the drawbacks of each project. This will show us what an ideal solution should be, and give us the basis for unifying the different projects, providing a series of improvements in the domains of application management and application analysis. At the end of the state of the art, we provide a view of what problems exist in current solutions, and give insight on what an ideal solution would entail.

2.4.1 Project summaries

2.4.1.1 DTrace

DTrace [Cantrill et al., 2004] adheres to a certain number of principals that make it feasible to be used in production systems, and is one of the main differences between DTrace and preexisting solutions like LTT [Yaghmour et al., 2000], DProbes [Moore, 2001] or Kerninst [Tamches et al., 1999]. These ideals are that a performance analysis infrastructure must have zero probe effect when

enabled, and must be absolutely safe. That is, its mere presence should not make the system any slower, and there must be no way to accidentally induce system failure through misuse. DTrace uses a high-level and safe instrumentation language that has been dubbed D. DTrace is platform dependent and tightly integrated with the operating system kernel, making it difficult to migrate to other platforms. Systemtap [Prasad et al., 2005] has been created for the Linux platform and can be seen, at least conceptually, as a Linux clone of DTrace (for the Solaris OS). Though the implementation is very different, it remains platform specific. Systemtap is proof that migrating such platform specific frameworks is a delicate matter. Furthermore, DTrace uses probes to insert code into running applications or the kernel. These insertion points or hooks are generally predefined and many even require modifying the C compiler in order to be inserted. This is precisely the case of the binary instruction "no-op" (no operation), which is an empty operation inserted at the start and end of every function. This instruction is then overwritten by DTrace with a method call into the DTrace framework in order to execute instrumentation. Although the no-op instruction does not produce any noticeable overhead even on micro-benchmarks, it is a static solution, not dynamic as DTrace would like to imply.

DTrace provides a manner of understanding applications and OS as one in order to help in systemic problems. It does not provide insight on how applications divide workload or how a single request is spread across different software components. A fine-grain analysis of workloads is required. DTrace does not provide mechanisms to modify the execution of applications, nor any support for non-functional concerns.

2.4.1.2 Project5

Project5 [Aguilera et al.,2003] infers dominant causal paths in distributed systems relying on tracing messages between nodes and using offline algorithms to infer causality from these traces. Two statistical algorithms for inferring causal paths have been proposed and have many disadvantages. First, they are statistical so no individual calls can be distinguished. They provide a very general vision of how components interact and which components cause bottlenecks. The algorithms are performed offline because they are costly and because they need a large amount of traces in order to correctly infer causality. The complete set of traces to be analyzed has to be obtained in one pass, because results from analyzing one set of traces cannot be aggregated with other results. Large amounts of traces are needed, causing large calculation times, which makes them not useful for online analysis. If during the time traces are being obtained the workload on the distributed system is altered or differs, the results will show a generalized analysis of the workload, making it difficult to understand the system or it may cause programmers to come to erroneous conclusions. Statistical algorithms only show relevant results on execution paths that are frequent. That is, a call path is going to appear as interesting only if it has been executed repeatedly with very similar execution times. The more variance within component execution times, the less exact the results of the algorithm will be, making it more difficult to understand the application. Also, aberrant or seldom occurring behaviors of an application are completely discarded.

Statistical analysis calculations are costly and it is only a "best-guess" calculation. Individual calls are not identified. We call Project5 a "best-guess" approach because correlation does not imply causality. Loosely interpreted this means that coincidence is not proof, and this is a basic limitation of both of their algorithms. Limiting the solution to after-execution is an important drawback because the information obtained cannot be used real-time in order to make immediate decisions on the execution of the application at hand, nor is it possible to provide meta-application support.

2.4.1.3 Magpie

Magpie [Isaacs et al., 2005] is a toolchain that helps understand system behavior by automatically extracting individual requests from a live system, and then constructing a probabilistic workload model from this data. The toolchain relies on instrumentation in the kernel, middleware and application-level components to generate events. Magpie is platform specific and relies on modifying communication channels that applications use in order to obtain useful information. These modifications are not dynamic and produce continual overhead. The solution is low level and relies on complicated event models tailored to the application to properly interpret the event produced. Even though offline and online versions of Magpie exist, they are only useful for analyzing the application, and cannot be used for meta-application or context propagation because of the time needed to parse events. Although non-functional concerns are not addressed by Magpie, the latter provides automatic workload analysis based on a request granularity, and this is the strong point of the project.

2.4.1.4 Causeway

Causeway is a general meta-application infrastructure for multi-tier applications. They provide an automated way of metadata propagation by instrumenting the kernel and kernel libraries. They provide an API for applications to insert when inter-thread communication occurs but does not pass through the augmented kernel nor through the augmented kernel libraries. Causeway has many disadvantages. There are a large amount of OS modifications required that cause permanent overhead for all applications that use operating system channels, and also make it platform specific. It is not fine-grained nor is meta-application interaction consistent. The granularity of meta-applications is restricted to interception points inside the OS. These interception points can have callbacks assigned to them, and are the basis of meta-application interaction. Applications that interact frequently with the OS communication channels have many meta-application interception points, but applications that do not communicate using the OS have a very limited number, making the meta-applications of little use. Causeway does not use a request granularity, and metadata that is added or modified can only be used by software *downstream.* Causeway does not support the analysis of an application, it only provides automated metadata propagation. No profiling techniques have been envisioned or are supported with the infrastructure. Causeway identifies communication points but cannot distinguish individual elements of the application, like the function that performs the call. Another limitation of Causeway is that a thread is considered causally dependent to only the last event or last entity that it has interacted with. That is, a thread only holds the metadata related to the last information read from one of the instrumented system channels, nothing more. This does not respect causal dependency, where one actor may be causally dependent to multiple other actors.

2.4.1.5 Defensive Programming Toolkits

Defensive Programming [Qie, 2002] suggests that a programmer must take an active role and provide systematic proactive protection against DoS attacks by embedding general mechanisms into software. The first approach is to insert the annotations directly into the code. Not only are annotations inserted, but the programmer specifies what actions are to be taken in case one of the conditions specified becomes true, causing a mix of non-functional and functional concerns that can only be changed by modifying and recompiling the application code. The solution is not dynamic, nor generic, nor does it support metadata propagation, meta-applications, or profiling of any kind. It is specific to DoS and resource consumption concerns.

The second solution [Schiavoni, 2006] is proposed for component-based systems and separates functional and non-functional concerns by specifying the annotations in component design files, instead of directly inside the application code. It has many of the same drawbacks as the previous toolkit, but improves annotations by making them modular and dynamic. The limits imposed on their work lie on their dependence to AOP. There is constant overhead if they wish to be runtime-dynamic. The annotations are used to specify services provided by the application and cannot be as fine-grained as individual instructions.

2.5 Remaining issues

There has been a large distinction between different areas in computer science, as is performance analysis, workload profiling, context propagation, and application management. Their particular objectives are not the same, but all these areas rely on the same basis. All of them require obtaining information from the application in order to perform their specific tasks. Even though they share this common trait (i.e. a need for instrumentation), they all utilize different granularities for achieving their goals. Profiling analyzes which functions in an application are being highly used or where bottlenecks exist. Workload analysis characterizes how applications react to different types of information it services. Meta-applications attempt separating non-functional concerns from the application, but they have been limited to interaction points between the OS and the application, making it difficult to have a true understanding of the application itself. This causes an application programmer or an administrator to be forced to analyze many different concepts and relate the results on his own between the different solutions he uses.

Application instrumentation is the foundation for these different areas. Ideally, instrumentation should be implemented as a dynamic infrastructure that provides variable interaction points that are adjustable when necessary, focusing on points of interest in the application. In reality, interaction points in existing solutions are rigid and cannot be changed. Existing solutions have been limited to a fixed and generally large granularity. Internal operations of an application go completely unnoticed. In many cases, only external communication is intercepted, and only these points permit application analysis and application management. Furthering difficulty for programmers, external communication points are not consistent, and can vary from one application to another. Some applications may provide many interaction points for meta-applications, while others do not. In short, meta-applications are not useful for applications they cannot interact with, and meta-applications must be tailored to the specific limits of applications that do provide useful interaction points.

In Table 1 we compare each project to the most important characteristics that would make for an ideal solution. The characteristics are separated into two groups: *analysis* and *management*. This division is sometimes not so clear do to the common dependency on *analysis*. It shows that solutions that provide good instrumentation techniques do not provide application management capabilities. Solutions that provide *meta* functionalities do not understand or instrument the application very well. This gap must be bridged in order to provide a unified view of an application. This unification requires a common granularity. The granularity must be equally useful for performing application instrumentation, workload analysis, profiling, meta-application construction, and developers.

		Main Objectives	Application Instrumentation and Application Analysis Characteristics[1]: Runtime Dynamic	Zero-overhead when not enabled	Platform independent	Fine-grained interaction or interception points	Application Management Characteristics: Provides execution interception[2]	Provides *meta-application* functionalities[3]
Projects	**DTrace**	Locate systemic problems.	Depends on probe implementation.	Yes.	No.	Yes, but depends on probe implementation.	No. Only obtains data from the system.	No.
	Project 5	Statistical causality for distributed systems.	Yes, when using passive tracing.	Yes.	Yes. Used for distributed systems.	No. Can only distinguish individual nodes, nothing else.	No. It is passive.	No.
	Magpie	Request tracking, workload profiling.	No.	No.	No.	No. Interaction limited to OS interception and application events.	No. Only obtains and analyzes information.	No.
	Causeway	Meta-application infrastructure.	No.	No.	No. But concept is implementable in other OS.	No. Interaction limited to OS interception.	Yes.	Yes.
	Annotation Toolkit (inter-code)	Intercept/modify execution.	No.	No.	Yes.	Yes.	Yes, but policies are specified before execution, and cannot be modified at runtime.	No.
	Annotation Toolkit (component-based)	Dynamic Annotations.	Depends on AOP implementation.	Annotations yes. But dynamic AOP causes constant overhead.	Yes.	Limited to method or function granularity.	Yes, but policies are specified before execution.	No.

Table 1: Comparison table of projects and desired characteristics for application instrumentation and application management.

1 Refers to application instrumentation, profiling and workload analysis.
2 Can modify execution of the application.
3 Provides high-level concepts permitting the construction of meta-applications.

Chapter III

3 Details of the Contribution

3.1 Overview

We have analyzed the state of the art and we have seen the limitations that exist. Solutions are specific to the exact problem they satisfy, and cannot be used outside of their particular context. Most solutions are not feasible in production systems because they are either not dynamic or they produce too much overhead. Many of the solutions proposed rely on the same bases, like software tracing, but even so they provide disjoint views of the application they study. Concepts have not been generalized and are difficult to interpret from one tool to another. Tools are specific for a specific task and developers are forced to bridge these conceptual gaps on their own. It is necessary to unite the solutions providing a fine-grained, high-level and dynamic solution.

We propose a system that improves on existing work by removing the limitations seen in the state of the art. The system is based on *Component-Based Software Engineering* (CBSE). CBSE is a branch of the software engineering discipline, with emphasis on decomposition of the engineered systems into functional or logical components with well-defined interfaces used for communication across the components. Components are considered to be a higher level of abstraction than objects, and as such they do not share state, they communicate by exchanging messages that carry data. Components are black-box entities that express their interactions with other components through well defined interfaces. Interfaces can be of two types, *client* or *server*. Server interfaces provide the functionality of the component, and client interfaces are used to express a functional requirements of the component. Client interfaces are bound to compatible server interfaces. Bindings are the interaction points between two components and can be modified at runtime. Components provide means of unbinding and rebinding their client interfaces to other compatible server interfaces. This is useful for constructing runtime dynamic applications, that can adjust to the requirements of the system by remodeling the application itself. Components can be added, removed or modified while the application is in execution.

Our system is useful for centralized component-based applications (i.e., single memory address applications). We have chosen an implementation based on the component model because of its modularity, its well defined interceptable interaction points and its runtime dynamicity. These three characteristics are key to permitting application instrumentation to be dynamic, fine-grained, and based on a consistent granularity. The component model provides functionality that existing solutions did not have and were not able to exploit, making them static and coarse-grained.

Our solution extends the concept of *request*, as used by web-servers, to component applications in general. The idea behind using requests is to represent application activity by regrouping activity to a single action that is initiated externally, instead of viewing application activity through single entities (e.g., single software component, thread). This is more intuitive for developers and provides an understanding of the application based on the services it provides. Requests are the base granularity for our solution. Requests are messages sent from a client interface to a server interface for treatment. A request can then be divided into smaller tasks, and serviced by different software components simultaneously. These task divisions are performed in order to achieve parallelization and optimal use of resources in the system. They are in fact, causal information pathways that exist in the application, and must be analyzed in order to regroup activity to its originating request. Furthermore, there is potentially no limit to the amount of requests being serviced by the application, so each component of the application may be servicing multiple requests at the same time. A request history is recorded during its servicing, and is called a *request execution path*. The request execution history includes important information (e.g. the components used to service the request, the time spent by each component, the task divisions and ramifications produced servicing the request).

The first problem we face is the construction of requests as a unique and universal entity for application analysis. This is achieved by instrumenting the application. Component applications are instrumented using *dynamic tracers* that analyze all inter-component activities. Dynamic tracers are inserted between bindings and produce events every time a message crosses a component boundary. Dynamic tracers are insufficient for request tracking because components are black-box entities, so causal information pathways inside a component go unseen (e.g., where requests are divided into subtasks). It is necessary for these pathways to be seen by the tracing infrastructure. This is achieved by instrumenting the component itself, turning it into a gray-box component. Per-component instrumentation is achieved using an annotation toolkit for identifying asynchronous execution. Dynamic tracers and the asynchronous annotation toolkit produce the information necessary, given simple analysis, for constructing requests. This provides us with online, fine-grain and deterministic request analysis.

Requests are now viewed as an entity in the application. We provide functionality based on requests. A request consumer interface is provided for applications wishing to perform workload analysis, although we do not provide application profiling ourselves. External applications can consume request execution paths and do application profiling themselves. Also, we provide meta-application construction using the same request granularity. Meta-applications are used for non-functional concerns in the application and are created separately from functional aspects of the application, improving the *separation of concerns*. Basically, a meta-application interprets the non-functional concerns specified by the user, and modifies execution of the application accordingly (see Figure 3 for a meta-application overview). Meta-applications require metadata in order to perform non-functional concerns. We group metadata into contexts and provide two unique context types, namely *request context* and *message context*. Contexts are referenced instead of propagated, avoiding the overhead involved in constantly copying contexts. Meta-application interaction points are called *callbacks* and performed by interrupting momentarily execution of the application. Callback interaction points are voluntarily limited to component interfaces, in order to maintain the separation of concerns and not disrupt internal functionality of the component[4]. This provides consistent, fine-grained (i.e. individual components are distinguished), and fully dynamic meta-application construction that benefits from requests as the workload division of choice. Finally, the solution is platform independent and can be implemented in many different component models.

4 Components are well defined and modular entities. It is not recommended to interrupt their internal functionality.

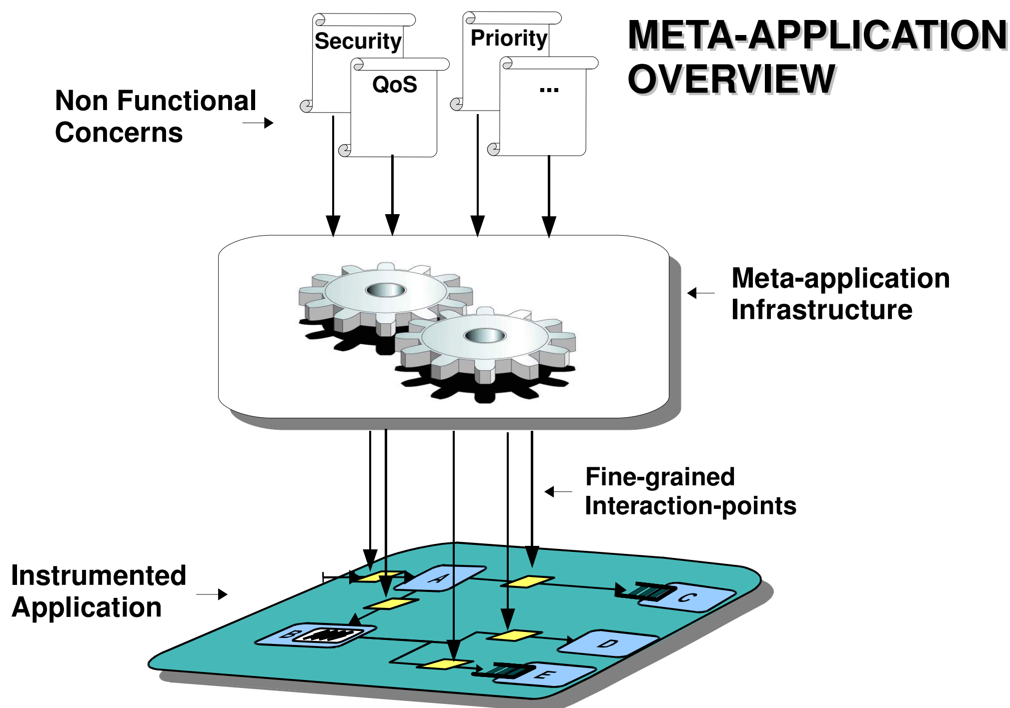

Figure 3: Meta-application overview.
Illustrates the concept of meta-applications and how they can interact with applications. Interaction is provided by fine-grained instrumented points in the application. The meta-application infrastructure must analyze the non-functional concerns and provide the meta-application functionality to the application.

In section 3.2 we explain how to monitor synchronous interactions between components using dynamic tracers. Section 3.3 details how we study asynchronous events using an annotation toolkit designed for that purpose. Section 3.4 describes how the information obtained from the instrumentation is used to automatically follow requests in the application. Section 3.5 describes the usage of requests for making metadata visible to all entities servicing the request. Section 3.6 explains the callback infrastructure and how it is used for constructing fine-grained meta-applications. Section 3.7 explains the request consumer interface used for profiling and workload analysis. Section 3.8 compares our solution to the other solutions introduced in the state of the art. Finally, section 3.9 is a summary of the chapter.

3.2 Synchronous interaction

As previously mentioned, in component-based software engineering, dynamic modifications are possible without modifying the original source code of the application. We rely on the reconfigurability of component-based applications in order to construct a tracing infrastructure that is dynamic. It can be inserted and removed at runtime and causes no overhead when not enabled. To construct the infrastructure we modify the structure of the application under study by adding new components. Primitive components remain black-box entities, because the tracing infrastructure does not modify primitive components and it is unable to know what happens inside a primitive component. As a reminder, primitive components are those which directly implement the functionality of the system.

In order to study how two components interact, we insert a tracer component in between the two components, permitting us to analyze each and every call that is made from the client to the server component. The tracer component is a simple and small component that does a basic analysis of the call that is being made. It intercepts calls from the client to the server interfaces, analyzes the call, and then delegates the call to the server. It obtains information like the interface the call is made on, the time of the call, thread ID, etc. More sophisticated tracers are also possible and they may analyze the arguments sent between calls and the results returned by the call, but for our purposes less complicated tracers are preferred. We use very simple tracers to minimize the effect of the tracing infrastructure on the application, which gives a truer understanding of the application since there is less interference.

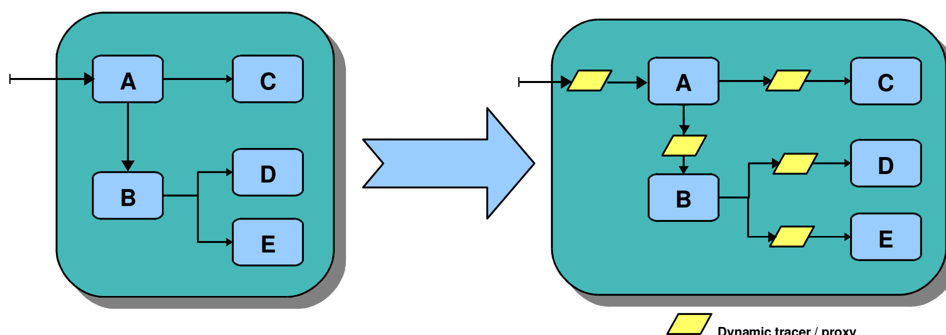

Figure 4: Dynamic tracers to instrument the application.
Left, shows a component application and interactions between components. Right, shows the same component application with dynamic tracers inserted into the bindings between the components. These tracers permit tracking thread execution paths in an application.

In Figure 4 you can see how a component application perceives tracers. The original application runs normally, then tracers are inserted into communication pathways of the application, and we obtain an extended and instrumented application. This tracing solution is a black-box solution respecting primitive components. This is because the internal functionality of a primitive component is not, in anyway, analyzed. Each tracer analyzes one bound interface of the application. Inserting tracers into every bound interface gives us information on all the inter-component interactions performed by the application.

It is important to understand how dynamic tracers function and what information is obtained. When a thread passes from the client interface of an external component, through the binding, to the server interface of the component under study, we say a call is being performed. This thread then executes instructions inside the server component, and when finished with the server calculations, it closes the call on the server component and returns through the binding to the client component. Each time the thread passes through the binding it is viewed by the tracer. This has a close resemblance to RPC calls in distributed systems, with the difference that the thread is followed using its threadID. This is basic functionality of a component call in a single memory-space. Calls performed in distributed domains are different and beyond the scope the this project.

Dynamic tracers provide a means of intercepting and halting an execution thread in the application. This is important because tracing statistics and actions regarding application execution can be performed while the component is executing. For example, if the calculations regarding the execution time of the call in progress exceed a certain threshold, it is possible to interrupt the execution and cancel the call. Calculations are thus performed in real-time and can be used for modifying application execution.

The precise creation of the tracer components is platform dependent, and varies from implementation to implementation, so it will not be discussed at this time. There are a series of steps which must be performed by all implementations of the tracing infrastructure, which are the following:

1. The Trace Infrastructure must be instantiated and started.
2. The application to be instrumented must be introspected and the instrumentation points located. These instrumentation points are bindings between interfaces.
3. For each binding to be instrumented:
    1. The components must be set to a passive state. Passive state is when no threads are executing inside the component. This limitation comes directly from the component model.
    2. One tracer component must be created. The tracer component must implement one client interface and one server interface of the same type as the binding being instrumented. Tracers have additional interfaces to interact with the tracing infrastructure (for more information on interaction between tracers and tracing infrastructure see Chapter IV Implementation).
    3. The binding to be instrumented is unbound. The client interface of the application is bound to the server interface of the tracer. The client interface of the tracer is bound to the server interface of the application.

In general, a tracing infrastructure of this kind is useful in applications that reside in a single memory space, but not for distributed systems. Further enhancements may eventually include distributed systems.

3.3 Asynchronous interaction

3.3.1 Overview

There are limits to the solution. Dynamic tracers can follow each thread through an application, knowing where a thread is, what components it has passed through, the time spent in each component, and many other useful elements. What dynamic tracers do not see are inter-thread communication points, because these events occur inside primitive components, which are not internally instrumented by dynamic tracers. It is necessary to understand and view asynchronous execution in order to properly trace applications.

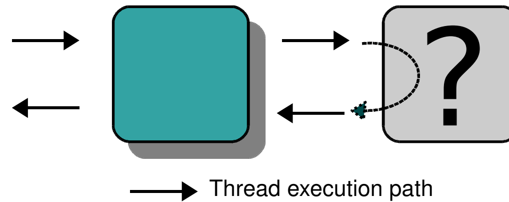

Figure 5: Hidden component functionality.
Shows an abstraction of thread execution in components. Components can be introspected, but the implementation code cannot be analyzed. Internal events that a component perform are not visible externally. The arrows show a thread enter and exit the component.

Figure 5 shows that there are limits on the information dynamic tracers can obtain. They study events that cross component boundaries, but are not capable of viewing intra-component events. This is important because it shows a need for finer analysis of applications in order to identify causal pathways.

Asynchronous execution is characterized by asynchronous events. A thread executes instructions one after the other in a sequential order. A thread receives data and then executes the instructions to service it. Data that is being serviced by a thread is in essence sequential, because it is associated with the thread that is performing the service. Events that break this sequentiality are known as asynchronous events. Inter-thread or inter-process communication are examples of asynchronous events. Asynchronous events are important to understand because they transfer information between different threads. This means that an original task can be partitioned and serviced by different threads. A common example is the partitioning of tasks in order to simultaneously service parts of it, making better use of underlying resources, like multiple CPUs. This is known as task parallelization.

There are many asynchronous events that occur in applications. These asynchronous events are performed for different reasons and causality between events is not easy to automatically identify. There are different ways of automatically identifying causality between asynchronous events in applications. The first is to make many assumptions, restricting the ways in which asynchronous events are carried out. These assumptions can be very limiting and in many cases they may incorrectly attribute causal paths, especially when the assumptions made are inaccurate. Whodunit [Chanda et al., 2007] calculates communication through shared memory automatically. They assume that all asynchronous communication through shared memory occurs inside of critical sections, and that the information written to the channel is calculated before the critical section, and information read from the channel is used after the critical section. Another approach is to have asynchronous events commented or annotated. This requires external intervention so the annotations can be correctly placed. Annotations can then be used to identify causal communication pathways generated by asynchronous events.

We analyze asynchronous events that a component application may perform and provide a series of generic information probes that are used to correctly attribute causal pathways. We will call these information probes *annotations*, but they are not to be confused with Java annotations or annotations in aspect oriented programming (AOP). The annotations are generic and implementation independent. They can be implemented in different platforms, under different conditions, with the necessary per-platform modifications and optimizations. Possible scenarios for implementing the annotations are different using technologies like Java-annotations, C-macros, an API, inter-code markers, trap instructions, etc. Their particular implementation will depend on the chosen platform, dynamic expectations of the application (e.g., compile-time, load-time, run-time), overhead of the annotation, and other conditions. Eventually these annotations could be

automatically inserted into code. Asynchronous communication points might be identifiable using static code analysis or executing instructions through a virtual machine like Qemu [Bellard]. This would be an interesting extension to our project, but for the moment we will suppose that a proficient programmer places them in precise locations in order to correctly exhibit asynchronous behavior.

The annotations should provide enough information for an external application to fully understand asynchronous events in the application under study. These events detect causal information paths. The causal paths are, in essence, request paths, since asynchronous events imply information passing. Figure 6 shows how annotations can be interpreted.

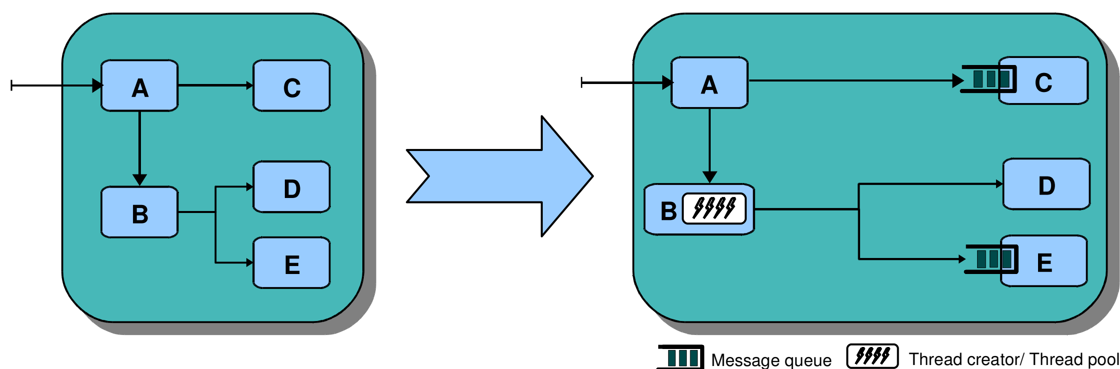

Figure 6: Annotation toolkit.
Shows an abstract view of the information that is obtained utilizing annotations. Left, an application without instrumentation. Right, the application instrumented with annotations, showing where and what kind of asynchronous events exist in the application.

3.3.2 Defining annotations

There are a series of annotations that may seem redundant in some cases because of the close resemblance between the asynchronous events. For example, message queues in shared memory may seem very similar to operating system pipes, or, socket communication may seem similar to port communication. The initial reaction is to provide a simple, reduced set of annotations as to simplify the programmers job of using them. This, in many cases is an optimal solution, but it limits an application from distinguishing between different asynchronous events. In applications it might be too costly to probe every type of asynchronous event at once, but it may be feasible to analyze them separately. Creating different annotations distinguishes events and permits us to "close" our view of events we are not interested in. Also, a specialized annotation may be more properly tailored to the needs of a certain asynchronous event, when a general information probe may obscure specific details.

We see two ways to approach annotation distinction. The first one is to create a highly specialized, quick execution probe for every possible asynchronous event. This limits growth of the infrastructure because a new event must be added to the infrastructure in order to accommodate new asynchronous events. This also complicates programmer productivity because of the ample gamma of probes to use and to choose from. The second solution is a general probe that utilizes different parameters so it can distinguish the different event types. This is simple for the programmer because he only has a very limited amount of probes to choose from, but probes are generally inefficient and not very expressive of the particular event. Our choice is a mix of the two former solutions. We have grouped asynchronous events into tight families and we add a couple of parameters to permit distinction between each family of events. Namely, our information probe

families are three: *thread calling, message passing, and data streams*. The additional parameters proposed are: `Label` and `Level`. These two parameters permit us to give a name to each individual event or to a group of events, and also to give it a priority or importance. The level attribute should be considered similar to debugging levels used by logging frameworks such as Log4j [Log4j]. These parameters are specific enough to permit a programmer to properly distinguish asynchronous events of interest with ease.

It is necessary for the information probes to not only identify when an asynchronous event has taken place, but also to obtain specific information from the application. For example, to follow causality in a message queue it is necessary to identify when the message is placed on the queue and when it is removed. This requires the use of message IDs. IDs are not limited to messages, and can be used for threads, message queues, ports, sockets, etc. An ID can be utilized for any entity of the application that requires it.

The information probes, and the solution in general, do not specify how to implement ID passing nor do they propose an infrastructure for doing this. In our implementation we leave it up to the programmer to support artifact IDs, for example, when passing messages it is up to the application to support the passing of the message ID. In many cases the implementation of IDs may be simplified thanks to the platform. Utilizing the same example as above, a message placed on a queue can use its memory address as its unique message ID, simplifying ID propagation. Avoiding the need to directly propagate IDs makes the tracing infrastructure as "light" as possible since calculations and storage of IDs are kept to a minimum. The effort invested from the programmer is minimal. In order to minimize effort for inserting annotations, the annotations could be inserted directly into underlying libraries in order to make asynchronous event handling transparent for applications that use these libraries. This is a solution chosen by Causeway [Chanda et. al, 2005] and SDI [Reumann et al., 2004] and has many disadvantages. For example, the granularity is not fine-grained and is limited to library calls. Also, the solution is static and affects all applications running, creating a constant overhead. Asynchronous events that do not use instrumented libraries go undetected. Finally, assumptions regarding how many messages are serviced by a thread must be fixed before hand, normally establishing that only one message is treated by a thread at a particular moment.

In general, the effort of inserting information probes is very small in well designed applications. Well designed applications should have asynchronous events wrapped in well determined method calls or functions that are easy to instrument. This means that even if an event occurs many times in an application, the function that causes it is generally written only once and has to be instrumented only once. Code modifications are necessary in very specific points, and this provides support for rapid and deterministic causal path detection.

3.3.3 Proposed annotations

The proposed annotations are for implementation in component models, but are not based on the particularities of any particular platform or implementation. The component model increases modularity and code decoupling and thus limits some forms of communication, like shared memory or shared variables. This is important because most implementations for thread synchronization rely on shared variables, thus forcing programmers to find new methods of implementation which are accepted in the component model, like message passing.

Some annotations may be seen as redundant since information may be directly provided by the platform. For example, Java does not distinguish threads as having a parent/child relationship, making all threads equal. Other platforms do provide this relationship, so the annotation for this

relationship would not be necessary. The annotations provided should suffice for an easy implementation in different platforms utilizing different technologies. Platforms that provide more information for asynchronous events provide a means for reducing the number of annotations used. Eventually, if enough information is directly provided by the platform itself, the annotations would be unnecessary, and all the asynchronous events could be automatically detected and analyzed. More general than the exact syntax of an annotation, the following annotations show what information is necessary for causality analysis of asynchronous events.

3.3.3.1 Thread creation and thread pools

Threads are system entities created to execute instructions. Threads belong to a larger entity, a process, where every process has at least one thread. We focus on single memory space, single process applications. In these applications, there may be multiple threads interacting throughout the system. In order to utilize a thread, the thread must be created or an existing thread may be utilized, for example, by calling a thread pool. To properly determine causality and follow requests in an application that creates threads or uses thread pools, we are required to know the ID of the calling thread, the ID of the called thread, the moment the asynchronous event occurs, and finally, in which component it occurs. This information is sufficient for causality analysis when we make a couple of assumptions. The assumptions we make are:

1. Each thread that is currently available has an individual, application unique, ID. After a thread dies this ID may be reused.
2. A thread is causally dependent to only one thread at a time, its caller. That means that a thread cannot be called to perform two tasks from different threads at the same time. In the case of thread creation, the execution path of the created thread is dependent of the thread that created it. For thread pools, the execution path of the called thread is causally dependent only of the last thread that called it.
3. Every task that a thread performs is causally dependent of the caller thread and is associated with the execution path of the caller thread unless clearly expressed otherwise (more details on causality decoupling in section 3.3.3.4 Independent execution).

Finally, we propose the following annotation for thread task delegation.

```
Thread_Called(Callee_Thread_ID,Called_Thread_ID,Type,Label,Level)
```

`Callee_Thread_ID`	The ID of the thread that performs the call.
`Called_Thread_ID`	The ID of the thread that is called and executes the delegated task.
`Type`	Thread Create or Thread Pool.
`Label`	Generally a string that relates a name of the particular information probe or of a group of information probes in order to distinguish them.
`Level`	A positive integer that indicates the importance or priority of the information probe. The lower the integer the more important the probe.

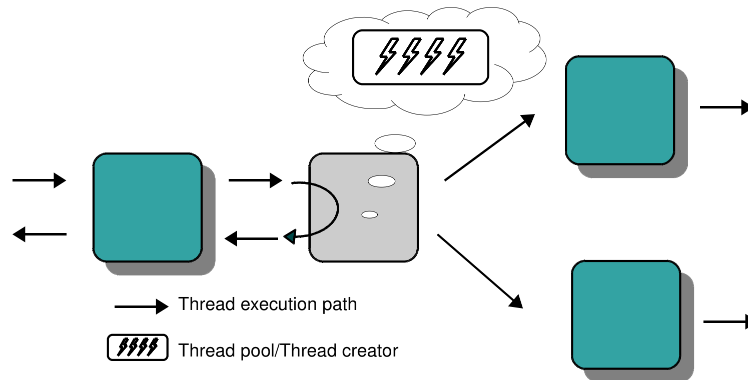

Figure 7: Thread pool/Thread creator.
Shows an abstraction of a component that uses a thread pool. Execution paths fork after the component into different components. The thread that is called is causally dependent to the thread that called it. Thread annotations make this information visible to external applications.

As shown in Figure 7, the `thread_called` annotation provide a means of identifying which components create threads or utilize thread pools.

Optimizations and simplification of annotations are possible when the platform provides additional information. For example, if direct support of thread relationships exists (e.g. parent/child relationship), it would not be necessary for programmers to insert the annotation that identifies thread creation. This particular optimization implies that the tracing infrastructure would not know exactly when a thread was created, but once the newly created thread performs a call and exits the component it was created in, it would cross through a dynamic tracer and then the parent/child relationship could be analyzed, creating the causal relationship.

3.3.3.2 Message Passing

Components communicate by sending each other messages. Messages are a unit of information utilized by the application. Messages are not necessarily the same size and how they are implemented can differ depending on the platform or the needs of the application. What is important is that messages are not shared across components. Once a message is sent, the component that sent it no longer has access to the information.

Message passing is not always easy to identify. Message passing should not be confused with data streams, even if some implementations confuse the two concepts. For example, a socket used by a web-server for incoming requests operates like a data-stream, but the information obtained is grouped into a single request that is treated as a message. That way, each request can be considered as a message passed from the client to the web server, or vice versa, hence we should use a message passing annotation in this case. Other cases are usually much simpler, as a producer-consumer application, where threads produce messages and place them on a queue for other threads to consume. Message Passing information probes are to be used when a distinct, well defined message is passed between two entities and that message can be clearly identified. We propose the following information probes for message passing:

```
Message_Sent(Message_ID,Entity_ID,Label,Level)

Message_Received(Message_ID,Entity_ID,Label,Level)

Message_Read(Message_ID,Entity_ID,Label,Level)
```

`Message_ID`	Must be a unique message identifier for that entity. Messages are placed on entities, so the message id must not be repeated for the specific entity.
`Entity_ID`	Must be a unique entity identifier for the application. This is used to uniquely identify message queues, sockets, ports, etc., being used for message passing.
`Label`	Generally a string that names the particular information probe or group of information probes in order to distinguish them.
`Level`	A positive integer that indicates the importance or priority of the information probe. The lower the integer the more important the probe.

`Message_Sent` is to be inserted in the code exactly before a message is sent to a message entity. `Message_Received` is to be inserted immediately after a message has been read from an entity. This removes the message from the queue making it no longer reachable from other threads. `Message_Read` is to be used carefully in the case a message is read from the queue, but is not removed. This is for cases when a thread reads the message and another thread reads the message at a later time. Both threads from the point the message is read, are causally dependent to the thread that has placed the message.

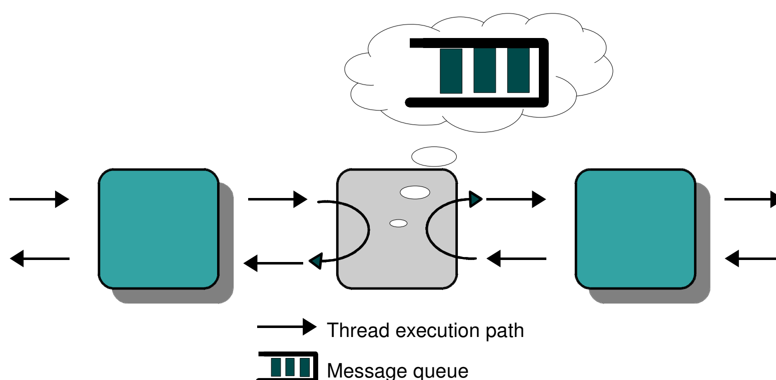

Figure 8: Abstraction of message queues.
Shows an abstraction of a message queue communication model. There are different components that communicate with a component that stores a message queue. Threads enter the component and place or remove message from the queue. Message passing annotations make these events visible to external applications.

Because we study one application at a time, and the application lays in the same memory space, a simplification to the message queue annotations is to utilize the message address as its unique identifier. This is only possible when messages are passed by reference, like most object oriented platforms do. Supposing no message address is repeated in the application, there would be no need for an `Entity_ID`.

3.3.3.3 Data Streams and Files

We identify another two types of asynchronous events, namely data streams and files. At the moment we do not propose annotations for these types of events because these asynchronous events are causal relationships. Annotations should be simple. In order to create simple annotations for these cases many assumptions must be made. We will still analyze the events and we propose using bit-ranges to stock causal relationships between readers and writers of the channel. Data

streams are incoming and outgoing streams of information. These streams have differences with message passing, since the stream cannot easily be packaged as a single message. This implies that there can be differences between the amount of information written at once and then the amount of information eventually read at once (see Figure 9), creating a complicated scenario of overlapping causality between writes and reads performed on a stream. Data streams generally imply a FIFO ordering on the bytes that are sent.

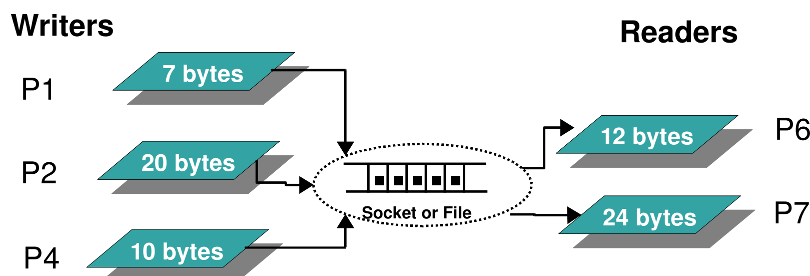

Figure 9: Multiple processes access data stream.

Shows an example of multiple processes reading and writing to the same channel. The amount of information can vary on every action and it is unclear which process has written information because the information is homogeneous. Causality tracking is complicated in these cases, but bit ranges could be stored for identifying causal dependency.

Files on the other hand do not have FIFO ordering restrictions. Files, on disk or in memory, can be long streams of bytes with reads and writes performed in different places simultaneously. A possible technique for identifying causality within data streams and files is stocking a reference to the bit ranges that are modified by a particular thread. As shown in Figure 9, reads and writes do not have to be the same length, one read could overlap several writes when consulting these bit ranges, hence the read is now causally dependent of multiple writes.

Generally in an application, data streams can be wrapped and interpreted as messages, limiting the necessity of calculating complicated causal nestings within streams. There are applications where it is not possible to construct messages out of data-streams, and this is why we consider it a different case to analyze. Files, in difference with data streams, are frequently used and pose a potential bottleneck for finding causal relationships within threads because of the overhead involved. Complicated stockings of information imply complicated methods for analyzing causality.

As a note, the same technique used on data streams could be used as an optimization when performing message passing. When one thread continually writes an extensive amount of messages, it would only be necessary to record the range of messages written and the thread who performed the write. Then, any reading thread would be causally related to the writing thread if his read is in the pertinent range. This optimization clearly improves memory usage and speed when one thread writes many messages continuously because fewer references are stocked and less calculations are performed.

3.3.3.4 Independent execution

In many instances automatic causality tracking can be too strict and extensive and may not correctly interpret the intent of the programmer. In these cases we provide causality-breaking information probes to indicate that the following actions performed by the thread after execution of the annotation are now independent of the previous actions.

Generally, this should be used as a correction on programmers intent. For example, in a web server a thread may enter a cycle of indefinite duration to read incoming requests and delegate the tasks on new threads. These tasks are requests from clients. In this case you have two threads, one to perform the read and one serving the request, and it is most likely the interest of the programmer to interpret each request as independent from the thread performing the request reading. With our earlier annotations, decoupling these events was not possible. In this case a causality-breaking annotation should be inserted in order to force decoupling between asynchronous events. The proposed information node is:

```
independent_execution()
```

This information probe is used to express that the actions performed before are now causally independent of the actions to be performed after. This is useful for isolating events or for forcefully decoupling causality to better interpret the applications intent, or the intent of the developer.

Causality decoupling provides a certain amount of improvements regarding memory and execution costs. To understand this, we must analyze the context that annotations are used in. Annotations are used for constructing request execution paths (more on this in section 3.4 on Request tracking). When causality is decoupled, smaller execution paths are recorded in memory. Since the execution paths are smaller, they will conclude earlier and can be freed from memory sooner than a large and complicated execution path. Also, having smaller and less execution paths saved in memory would improve execution times because less comparisons and less searching is necessary to create the execution paths.

3.4 Request tracking

3.4.1 Overview

The term *request* is used by web-servers, but it is not limited to that domain. We apply the term request to a message sent from a client interface to a server interface of a component for treatment. These messages are serviced internally by the component. A message can be divided into smaller tasks and serviced by more than one thread at a time. An application may service multiple requests simultaneously, and it is not uncommon to have multiple threads executing inside the same component, complicating request tracking.

We propose automatic request tracking for multi-threaded component-based applications. Request tracking correlates events produced from dynamic tracers and from the annotation toolkit to their originating request. Dynamic tracers provide events regarding thread execution. These events are obtained at component borders and specifically are events regarding calls on component interfaces. The annotation toolkit produces thread communication events that occur inside components. Request tracking performs the analysis of these events and maintains a per-request record of the execution path and of the ramifications of that execution path. In essence, request tracking identifies and records causal information pathways in the application. Basically, all activity in the application can be related to a message sent to one of the services provided by a component.

3.4.2 Request execution paths

Request paths are the execution history or execution path of all threads that have serviced the request. They detail which components were involved in the treatment of the request. A request path can be interpreted as the call graph created from the servicing of a request, which may include multiple thread execution paths with asynchronous event links to unite them. Individual thread execution paths are associated with each other using asynchronous events, producing a complete, per-request, call graph that represents the components used for treating the request (see Figure 10).

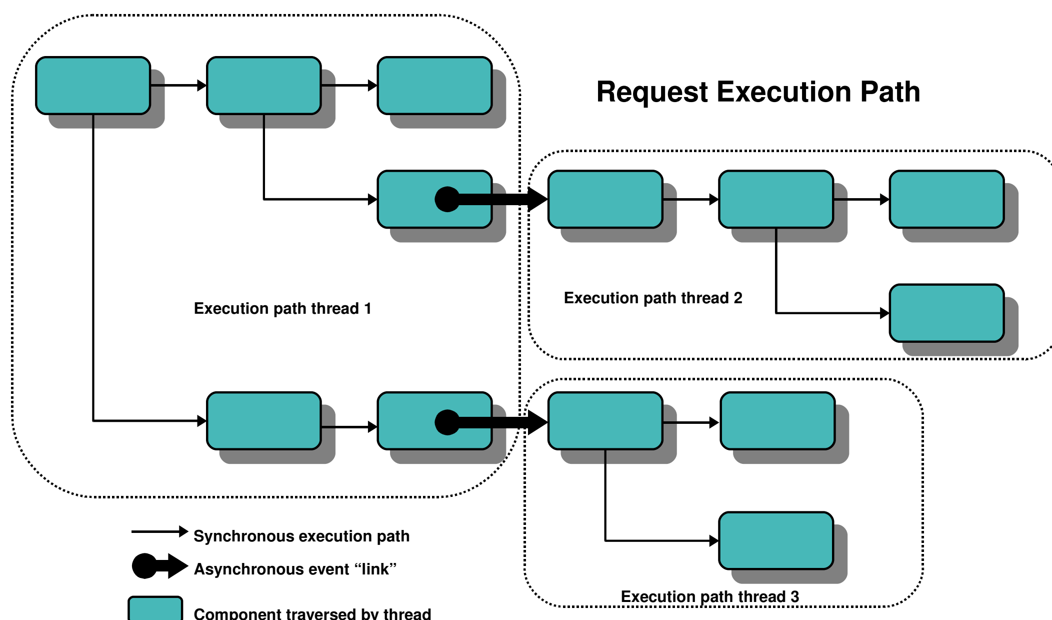

Figure 10: Request execution path.
This figure is an example of a Request Execution Path. A request execution path shows the full list of components traversed by the threads that serviced the request, and their inter-component interactions. The component name and the time a thread has spent inside each component are recorded. The request path is composed of individual thread execution paths, that are linked utilizing asynchronous events. The thread execution path is determined using dynamic tracers. Links between thread execution paths are created using the asynchronous event toolkit. A thread execution path can be compared to call graphs used commonly for profiling legacy applications.

Dynamic tracers provide information to construct the execution path of each thread. They are capable of threads when they cross component boundaries and they provide enough information to create a per-thread execution path. Requests can be serviced by multiple threads. When a request of a service is made, there is only one thread that initiates treatment of the request, but asynchronous events provide a means of increasing the amount of threads that service a request. These asynchronous events are seen utilizing the asynchronous event toolkit.

A request initiates with one execution path, because there is only one thread that begins servicing. While servicing the message, an asynchronous event may occur causing a split from the original execution path. This creates two separate execution paths that service the same request. The asynchronous event serves as a union between these asynchronous paths. We utilize the asynchronous event toolkit for creating these links because the toolkit provides information on causal information pathways in the application, which is the pathway the message in service is taking. The asynchronous event gives information concerning the thread ID that has caused it, in what component it has occurred, what kind of event it is, and at what time. In other words, we follow the path that the message in service takes, creating a branch when information is sent from

this thread to another, and we record a history of the components involved and the time spent per component. The end result is a series of "glued" together synchronized execution paths, using asynchronous events as the glue.

At the moment, only the amount of time a thread spends per-component is recorded, and the amount of time a message waits on a message queue. Time is information that is easily calculated across different platforms. Resource consumption is not easily calculated. In order to estimate per-request resource consumption, like CPU cycles, network resources, disk resources, it is necessary for a low level event system to be utilized. This event system must provide information on the resources consumed by each thread. With that information it is only necessary to consult what request is being serviced by the particular thread and add the resource consumption to that request. It is not our interest to work on an event system because they are platform specific, and not portable. Resource consumption can be rapidly added to the request tracking mechanism if such an event system exists.

3.4.3 Modifying request tracking granularity

The granularity of request tracking is provided by the dynamic tracers. Tracers provide the request tracking mechanism with information regarding the boundaries of components and when these boundaries are crossed. In order to distinguish every component in the application, it is necessary for a dynamic tracer to intercept every call between components. This causes overhead, and in some applications this might not be acceptable. If coarse grain tracing is acceptable or preferred, an application can reduce the amount of dynamic tracers, causing the request tracking mechanism to view less components. The amount of threads that are viewed by the dynamic tracing infrastructure would be the same, because the annotations themselves would not change, but the amount of components recorded in the request execution path would be less. Per-request statistics would remain the same, but since less components are distinguished, the per-component statistics would increase because the data from missing components would be added to components that are correctly distinguished, thus compensating for the missing components. Careful attention should be paid, because incorrectly placing dynamic tracers could cause the request tracking mechanism to miss requests being serviced. This is because a request is seen by the infrastructure when the first thread that services the request passes through a dynamic tracer.

3.4.4 Request consumer mechanism

The request granularity of a workload has already been proposed for workload profiling in web servers. We propose utilization of requests for profiling component based applications. This granularity better serves profiling applications, and helps in improving application performance and in debugging applications.

Our application does not provide any workload profiling nor does our work focus on profiling techniques. We do feel a need to respond to the necessity of workload profiling, so we have provided a request consumer interface. External applications can subscribe to events produced by the interface in order to perform application profiling. Each time a request is completed or is terminated, the request tracking mechanism sends the request execution path to all subscribers. The subscribers are free to analyze the request using any technique they implement. At the moment profiling is limited to temporal statistics because we have not provided an event system for resource consumption. This event system would provide workload profilers with much more information for characterizing the application under study, but event systems are platform specific.

3.5 Context propagation

3.5.1 Overview

Metadata is non-functional application data that is associated with functional application data. Metadata is handled externally from functional data, providing a separation between functional and non-functional concerns in an application. This separation promotes component reuse because different applications can use components in different non-functional contexts, or vice versa, the same non-functional concerns may apply to different components. It is our interest to permit an application to transparently and separately manage metadata by creating a meta-application infrastructure. For this to be feasible we must correctly deduct causal relationships between the activities that occur in an application. Our dynamic tracers and our information probes make causal information paths visible, providing enough information to correctly propagate context throughout an application. These causal pathways are analyzed by the request tracking mechanism. We propose requests as the application entity to be used for metadata propagation.

Request tracking provides the basis for our automatic metadata propagation infrastructure. We believe that metadata should be managed utilizing a request as the base granularity. This continues our idea to represent application activity at a per-request granularity. Metadata is grouped into contexts and propagated along side application data. Context is the base unit for non-functional data propagation. A context is associated with an entity in the application, being either a message, a request or a thread. A context must follow causal pathways as does functional data, maintaining this association throughout the treatment of the request. We propose two novel types of context, *request context* and *message context*. We provide an automated mechanism for propagating context through an application, and a callback mechanism in order to access and modify contexts. Our automatic propagation mechanism relies on request tracking and fully respects causal information pathways. Contexts are accessed using callbacks. A callback is additional functionality that is added at component frontiers. Callbacks are associated to dynamic tracers, because tracers interrupt component calls at component borders. Dynamic tracers halt a thread at the boundaries for two purposes; one, real-time construction of the request path and metadata propagation, and two, execution of meta-applications by means of callbacks. Context propagation is performed automatically and transparently in regards to functional components of the application.

3.5.2 Metadata key-value pairs

Metadata is a key-value pair that is saved in a context. Contexts contain a group of metadata and can be per-request or per-message. (Details regarding message contexts and request contexts are given later.) Each of these types of contexts records metadata in the same fashion, but they are propagated differently. Metadata can be added to a context, removed from the context or modified.

We propose two functions for treating metadata. These functions are simple getters and setters for metadata key-value pairs. It is important to note that since there are two different contexts associated with an entity at a time, we propose a separate function for treating each type of context. The first type being the request context, and the second the message context, which will both be explained later.

```
get_message_metadata(Key) returns Value

get_request_metadata(Key) returns Value
```

```
set_request_metadata(Key, Value)
set_message_metadata(Key, Value)
```

The `get` functions search for metadata that matches the provided key. When found, the metadata value is returned. If there is no metadata matching that key then a null value is returned. The `set` functions add metadata to the context and modify existing metadata by means of overwriting it. If there is no metadata matching the key provided then the metadata is added to the context. If there is already a matching key, then the metadata is overwritten with the new value.

In general, these `get/set` functions provide the necessary, but minimal, functionality for administrating metadata in an application. Eventually these functions could be extended to include much more metadata functionality or to optimize certain operations. For example, if a user wishes to update a value he must first perform a `get` to view if the metadata exists and to retrieve the metadata value, then perform a compare if necessary, and finally perform a `set` to insert the value. For the time being our `get/set` functions are sufficient for our purposes of constructing a meta-application infrastructure.

3.5.3 Request context (global context)

Request tracking, as explained earlier (see section 3.4 Request tracking), records information regarding which thread is servicing which request. This information permits us to relate a request context to all entities servicing the request. This context can be seen as a global context because there can be multiple threads or messages that reference it (see Figure 11). If the request context is modified, either by adding, removing or changing its metadata, these modifications will be instantly seen by all entities of the request. Request contexts are propagated through causal pathways. A request context, when created, is associated with a thread. When an asynchronous event occurs, the context continues its association with the thread that caused the event, and is now also associated with the entity that is causally dependent of the thread.

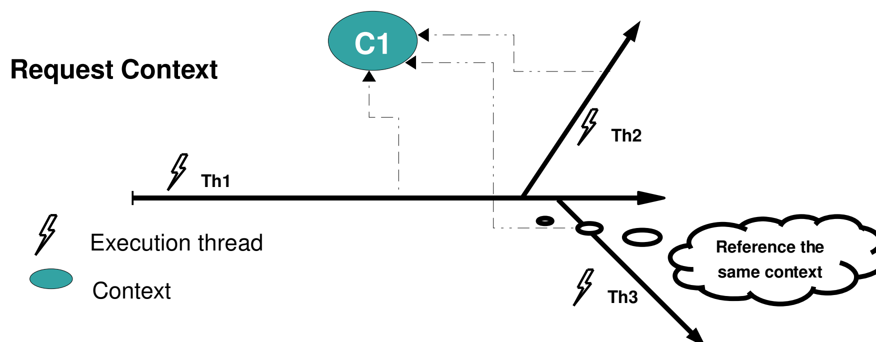

Figure 11: Request context.
We show how request contexts are referenced and how they propagate across asynchronous events. At each asynchronous event, the request context remains untouched. The dependent entity now references this request context, providing shared metadata between entities pertaining to the same request. Modifications to the context are viewed by all entities of the request

Some of the uses for a per-request context are security, priority, QoS, etc. A request can save metadata related to non-functional concerns of the application. This metadata can be accessed and will affect every entity of the request. An example of utilization is resource limits. For example, requests are permitted to consume a limited amount of resources. When servicing of a request starts, the `resource` metadata is set to zero. When servicing continues and resources are consumed, the request context is updated. If the request exceeds the permitted limit, the request

can be canceled. Cancellation is not directly supported, but threads can poll to view resource consumption and cancel execution if limits are exceeded.

3.5.4 Message context (local context)

A per-message context is related to the message being treated or stocked. Per-message contexts are stored within a thread context up to the point where an asynchronous event happens. At asynchronous events the context is copied and an independent, but identical context is created and stored with the asynchronous event (see Figure 12). A modification of the per-message context after an asynchronous event is local and does not affect the original or any other context. For example, if a thread places a message on a message queue, the context of that thread is saved until the point the message is received, and the context is then added to the thread-context. Modifications are only viewed "downstream", since the context is propagated and duplicated after these modifications happen.

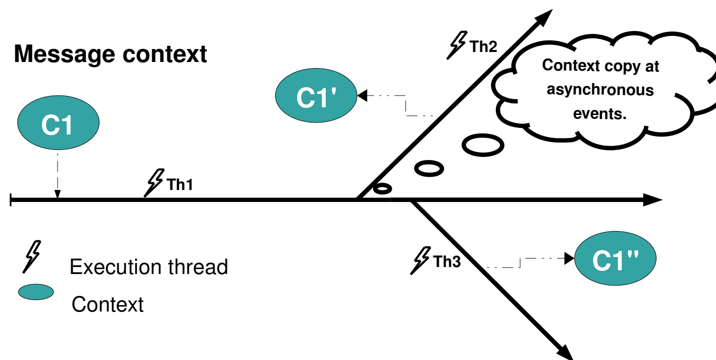

Figure 12: Message context.
We show how message contexts are referenced and how they propagate across asynchronous events. At each asynchronous event, the message context of the entity that initiates the event is duplicated. Each context is now independent and propagated separately through the application. Modifications to metadata in a particular message context are not visible to other contexts.

There are a series of uses for per-message metadata. Message contexts are used when a metadata is provided that is required at specific points later in the execution path. For example, if we analyze a two stage web-server, where the request is divided into two parts, the first one serviced by the stage that controls the dynamic content, while the second part is serviced by the stage that controls database access. If we were to control resource consumption, we can provide independent limits for each stage. CPU usage for the first stage would naturally be higher, than that of the second stage. Accessing the database implies disk usage, so the disk-resource limit would be higher in the second stage. With message contexts, the metadata type could be repeated across the application, with different values, and modifications to metadata are local. Other examples regarding priority or quality of service can also be imagined.

3.5.5 Handling multiple contexts

A thread can be causally dependent of different asynchronous events that, in essence, create a dependency link between various execution paths. Since these execution paths may reference contexts, the natural thing to do is now have dependent thread reference these same contexts because of the causal dependency. This is explained earlier where message contexts are duplicated and are specific to causally dependent entities and request contexts are associated by causally

dependent entities. We have not evaluated, until now, the possibility of a thread referencing multiple contexts because it is dependent of multiple execution paths.

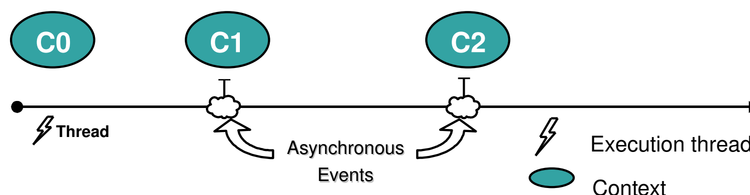

Figure 13: Multiple contexts.
We show that multiple contexts can be assigned to a thread. This is normally caused from multiple asynchronous events, of which this thread depends (e.g., reading messages from message queues). In order to respect causal dependencies, the contexts must be each taken into consideration, not only overwritten by the newest context.

To better understand the problem we shall start with a simple use-case. For example, a thread may read two messages from a different queues in order to execute a particular task. Supposing both messages reference a message context and a request context, we must analyze how to manage multiple contexts of the same type (see Figure 13). The following is a list of examples that must be dealt with and have not been considered by other solutions because existing solutions have limited causal dependency to only the last causal event.

- You may erase older contexts and replace them with newer contexts, limiting the amount of referenced contexts of each type to only one. This solution is used by Causeway [Chanda et. al, 2005] and SDI [Reumann et al., 2004], although they only reference one type of context each. It is chosen because they both assume that a thread treats only one message at a time. This means that the thread is causally dependent to only the last asynchronous event, which is often untrue and is also too limiting in many cases.
- You may have a thread reference various contexts of each type. The difficulty here is now related to providing a straightforward access to the metadata. It may be difficulty to expect a user to completely understand and know how many or what contexts are related to the specific entity at one precise moment. In fact, it is our supposition that a user does not know all causal dependency at a specific moment, as a part of our contribution is to provide an infrastructure to improve his understanding.
- You may create a union of contexts, overwriting only metadata that is repeated. When an entity is dependent of two contexts, you analyze each context and create only one, enlarged, context, containing each type of metadata without repeating. This would imply overwriting metadata values of types that our duplicated. This is a mix of the two former solutions because most elements of each context are preserved, but we eliminate repeated information. In fact, if a user accesses a context and expects a particular type of metadata to be there he will find it, although it might not contain the value he would be expecting to see.

In order to make things as clear as possible for a user we must try and make context modifications as local as possible. We have slightly different solutions for each type of context. In the case of two message contexts that become related to the same entity, we unify them into one, creating an enlarged context. We perform this feat by overwriting repeated types of metadata with the value inside the newer context. This solution is local since a message context is unique to an entity, only affecting events that follow “downstream”.

The previous solution works well with message contexts, but it is unacceptable with request contexts, because a request context can be referenced by many entities at a time, and mixing request contexts by overwriting repeated metadata types can modify other entities expected behavior, not to mention making it very complicated to understand where a metadata value actually came from or why it was modified. We this in mind, we wish to make metadata modifications local, since multiple dependency, which is the factor that causes multiple contexts, is also local. Our decision is to have an entity relate to more than one request context, saving the precise order of relation. This makes newer referenced request contexts more important than older ones. As before, we fall into the problem of handling repeated metadata types because two contexts can have the same metadata type. The solution, which is simple and can be eventually modified, consists in searching for the first metadata type that falls under the users search criteria. This makes the order in which the request contexts were saved important, since the newest reference added will be the first request context searched. To treat modifications to metadata values or adding new metadata key-value pairs we follow the same philosophy of newest context first. This means that new metadata is added to the newest referenced context and that modifying existing metadata is done by searching the contexts for the metadata type from newest to oldest until found.

Treating multiple contexts is complicated and our solution is by no means perfect. We provide a solution based on our suppositions of what a programmer would expect. There are cases that we can imagine where our solution is by no means the preferred one. Also, we should not forget that these entanglements of causally dependent information are less common than regular causal dependency, but there are many applications where they do occur and it is necessary for them to be treated according to the programmers intent.

Eventually, we could provide many different multiple context treatment methods and have the user specify which one he would prefer to use, or to propose his own management system. For example, a context priority could be used to specify that metadata in one context is more important than metadata in another, enabling priority metadata to overwrite other metadata. For the moment our solution is sufficient in providing automatic causality-respecting context propagation. Further pursuing the subject is beyond our immediate scope.

3.6 Callback infrastructure

3.6.1 Overview

A callback is executable code that is passed as an argument to other code. Usually, the code is passed as a pointer or handler to another function. In this case when a certain event happens or a particular piece of code is reached, additional or user described functionality can be executed. In component applications, callbacks, are themselves, implemented as components. Callbacks components implement non-functional concerns of an application. We propose implementing non-functional concerns at component borders, as to disturb as little as possible component functionality, providing a means to modify non-functional concerns at runtime and to increase modularity and component reuse. Callback interaction takes place when a thread enters a tracer. The tracer that is executing will notify the tracing infrastructure of the event. These events include, but are are not necessarily limited to, the following events:

- A server interface method is about to be executed, namely a "pre" method.

- A server interface method has finished, namely a “post” method.
- An error has occurred during the execution of the server interface method.

The tracing infrastructure must provide an client interface that notifies the callback manager of the event. If there is a callback associated with the event and the tracer in question, the callback is executed. Callbacks have access to application metadata. A callback is permitted to access and modify metadata associated with the information that the thread which executed the event is servicing. Operations on both, request context and message context, are permitted, but are limited to the contexts associated with the information being currently serviced. Callbacks cannot modify other request or message contexts.

At the moment we propose no limits on code that is executed by callback components, so special attention must be paid in order to avoid crashing an application or modifying undesired parts. Safe execution of callbacks could be supported in later versions.

3.6.2 Callback components

A callback component should be used for accessing, adding, removing and modifying metadata. Callbacks can perform decisions regarding execution of the application based on metadata values. This extracts meta-application behavior from the application, and it also makes meta-applications dynamic, since the callback components can be added and removed at runtime. In general, callbacks are used for all non-functional concerns an application may require, such as QoS, security, prioritizing requests, resource consumption analysis, etc.

Callback components should be created using a callback component factory. The idea is to limit the need for a user to be disturbed with the implementation of the meta-application infrastructure, and in particular with the callback infrastructure. A component factory creates a component that corresponds to the functionality that the user describes for each of the events supported by callbacks (i.e., pre, post, error). Then the component can be used and added to the meta-application infrastructure to interact with the metadata of the application. As a note, adding the component into the meta-application is not the same as defining meta-application interaction points. Meta-application inter-action points are described next.

3.6.3 Defining callback interaction points

Callback interaction points must be provided by the user. The callback components themselves are housed in the meta-application infrastructure, but the interaction points of callbacks are directly associated to dynamic tracers, because dynamic tracers interrupt normal execution at component boundaries. A user of the system then then specifies the interfaces in the application where a callback component should interrupt normal execution and provide additional functionality. The meta-application infrastructure then associates the interfaces provided to the dynamic tracers intercepting calls on the interface. There can be more than one callback associated to the same interface. This gives the user the possibility of generalizing meta-application behavior to different parts of the application, instead of having to add functionality one interface at a time. In essence, this means that one callback component can be called, possibly simultaneously, from different parts of application code. When selecting where to implement callback components, there may be overlaps with other callback components. This provides extra functionality and differentiates callback behaviors, encompassing them into independent objects. Callbacks must be prioritized because callback components may modify metadata, altering posterior callback component

behavior. Components with higher priority are executed first, and components with the same priority are executed in order of assignment to the interface.

3.7 Profiling

Profiling is one of the purposes of creating our infrastructure, but it is not directly performed by our infrastructure. We believe that a request granularity is useful for analyzing workload and application performance for all component applications. We provide a profiling interface, where profiling clients can subscribe to request execution path information. Each subscriber receives the request execution path of a newly completed request. The execution path includes, at the moment, all components traversed, all threads used to service the request, the asynchronous communication points and the amount of time each thread has spent in each component.

Resource consumption is also important to follow. At the moment we do not address this issue, because resource consumption requires a low-level event system and is fully dependent of the platform. The events required by our infrastructure for analyzing per-component and per-request resource consumption must indicate the amount of resources consumed by each thread. The meta-application infrastructure would then be able to add these resource consumption events to the appropriate request execution path record. Utilizing event systems like Event Tracing for Windows or newer propositions like JSR-000284 Resource Consumption Management API, would provide our infrastructure with the necessary information to add this functionality.

3.8 Comparison to other projects

We have compared our project and contribution to the projects that we studied in the state of the art. This comparison will show the differences between our work and existing solutions. We shall emphasize our advantages over the existing solutions.

3.8.1 Comparison with DTrace

DTrace [Cantrill et al., 2004] focuses on understanding how applications and the operating system interact as one. DTrace is implemented specifically for the Solaris operating system. Our project is focused on single component based systems in general, not any particular platform. We provide a general mechanism that can be implemented, although with small modifications or platform specific optimizations, on many different platforms and in many different component models. As such, our solution is platform independent, unlike DTrace. Systemtap [Prasad et al., 2005], a DTrace clone for Linux, has shown just how difficult it is to migrate the DTrace solution.

DTrace's instrumentation uses probes to insert code into running applications or the kernel. These insertion points or hooks are generally predefined and may even require modifying the C compiler in order to be inserted. Many of the probes are by far dynamic, and are introduced into applications by providing hacks. Our solution relies on the intricacies of the component model to provide dynamic instrumentation. Also, we provide request tracking, context propagation and application management. DTrace avoids modifying execution of applications in order to maintain a *completely safe* policy.

3.8.2 Comparison with Project5

The algorithms proposed by Project 5 [Aguilera et al.,2003] have many disadvantages. First, they are statistical so no individual calls can be distinguished. They provide a very general vision of how components interact and which components cause bottlenecks. The algorithms are performed offline because they are costly and because need a large amount of traces in order to correctly infer causality. Traces are a whole entity that are used to perform the calculations, that is, you can not calculate causality on one set of traces and then add another set of traces at a later time. Because the amount of traces is large, calculation times can be long, which makes them not useful for performing real-time decisions on application performance. If during the time traces are being obtained the workload on the distributed system is altered or differs, the results will show a generalized analysis of the workload, making it difficult to understand the system or it may cause programmers to come to erroneous conclusions.

In comparison with our work, Project5 is much less invasive but is performed postmortem and is only a heuristic. Our work is deterministic and much more fine grained, since we distinguish single components inside of a larger application, instead of individual nodes of a distributed system. Statistical analysis calculations are costly and it is only a *"best-guess"* calculation, and individual calls are not identified. We call Project5 a "best-guess" approach because correlation does not imply causality, and this is a basic fault of both of these algorithms. Also, aberrant or seldom occurring behaviors of an application are completely discarded. We can, on the other hand, analyze a single petition or request in the system, distinguishing every component utilized to service the petition and every thread used, making it possible to study and analyze application behavior in a fine grained and real-time manner.

3.8.3 Comparison with Magpie

Our solution is of a higher level than Magpie [Barham et al., 2003] because we focus at the component level. We avoid platform dependent solutions, like resource consumption analysis, which is based on low-level event infrastructures that would depend on the specific implementation of the component model. Instead, we insert dynamic interceptor components inside the application to study it, making the solution feasible for any implementation of the component model, supposing they support dynamic reconfigurations. Magpie does not modify application code but must instrument communication channels and utilize event libraries to perform request tracking. Our project, at the expense of inserting inter-code annotations which we feel is a minimal and simple effort, obtains a correct request path with low cost calculations and is a completely deterministic effort. These simple annotations remove the necessity of having complicated event schemes which themselves are error prone to produce and require a certain expertise and understanding of the application.

Magpie performs modifications to communication channels which cause permanent overhead on the running applications. These modifications even affect applications that are not being studied. The modifications are application specific and need to be modified for every application that is studied on the system. The models used to perform partial joins over the events in order to construct the request history are completely dependent to the application, and can not be reused. Our work requires a small effort to insert annotations into the application under study in order to correctly view asynchronous events and reliably construct request paths. Other than the annotations, request tracking is fully automated and completely dynamic, making it feasible to use in production environments.

Finally, our infrastructure is a meta-application infrastructure used for component models and also based on the component model itself. It runs parallel to the applications it analyzes, and does not affect other applications in the system.

3.8.4 Comparison with Causeway

Causeway [Chanda et. al, 2005] is used on multi-tier applications and thus, is very large grained in their meta-application interaction points. Causeway's static modification of the kernel and kernel libraries causes overhead on all instrumented system calls, even when Causeway is not in use. Our solution is dynamic and the instrumentation is application centralized, instead of OS centralized. The instrumentation does not affect other applications and can be inserted and removed even at runtime.

We provide a unified granularity for both, application management and application analysis. Causeway has focused on controlling applications only at external communication points. This is large-grain and not useful for great variety of applications. We provide fine-grain interaction and interception points in the application. We provide novel types of metadata, namely request and message contexts. Causeway does a simple *copy/paste* of metadata on communication channels. Another limitation of is that a thread is considered causally dependent to only the last event or last entity that it has interacted with. That is, a thread only holds the metadata related to the last information read from one of the instrumented system channels, nothing more. We suggest that a thread should be dependent to all events that have occurred before, unless specifically specified by the programmer. Because we respect causal dependency, we have provided methods for treating multiple contexts, and for decoupling causal dependency. Our solution is based on programmer's intent.

Causeway's choice to piggy-back metadata on threads causes a constant overhead which eventually limits the size of metadata. Every system call implies a transfer of metadata from the user-level thread to the kernel-level thread. Our solution saves references to metadata so when an asynchronous event occurs, and only when asynchronous events occur, message contexts are duplicated, because they are entity specific, but request contexts are referenced. These operations only happen when information is passed, not at every system call. Unlike Causeway, there is no threat of surpassing the overhead limit when associating new request contexts.

3.8.5 Comparison with Defensive Programming

We both utilize a set of inner-code annotations to provide us with necessary information. Although the information obtained and the way it is used is very different, the basic idea that programmers should take an active role in activities other than functionality, for example defense or tracing, is the same. The annotations used for defensive programming describe resource consumption, while our annotations describe asynchronous events. Resource consumption annotations not only obtain information, but they affect the control flow of the system. Control flow is specified by the annotation and is specific to DoS defense. Our annotations are simpler and do not change the execution of the program[5]. There are no extra control functions added, just a quick evaluation of the event in order to correctly construct the request path. Controlling execution in our solution is dynamic, fine-grained and done automatically at component borders. We provide a means of creating abstract meta-applications.

---

5 Annotations do not modify execution, but our meta-application infrastructure by means of callbacks does provide a means for controlling and modifying execution flow.

3.8.6 Comparison with A Posteriori Defensive Programming

The most important aspect of their work is that annotations can be applied after the design and implementation of a component and without modifying neither of them. It is possible to add DoS protection to an already deployed application using AOP techniques. This gives way to the a posteriori naming of the approach. The main defect in their work is the utilization of an annotation parser at runtime. It causes a constant overhead that we feel unacceptable. This is mainly because of their dependency on AOP technology, and the current limits that exist in AOP. Our annotations must be inserted into functional code, as they cannot be expressed separately because of the fine granularity required. This limitation has forced us to look at other options. We obtain dynamicity utilizing various technologies, namely a binary code parser (ASM [ASM]) and a Java tool (JVMTI [JVMTI]) for code hot-swapping at runtime. This solution is more difficult to implement and depends much more on the platform at hand, but provides a fast manner of activating and deactivating instrumentation. This platform dependency of our particular solution does not limit similar techniques being implemented in other component models.

3.9 Summary

In this chapter we have presented the necessary techniques and information for automated request tracking, context propagation and meta-application functionality. We present dynamic instrumentation in component-based applications by means of dynamic tracers. We give a thorough analysis of asynchronous communication events that exist in component applications and how to follow causal information paths through the application. We have paid special attention to correctly interpreting causal pathways in the application, and to providing dynamic instrumentation and interception points. We also have proposed two novel forms of contexts, namely message contexts and request contexts. The solution provides unified fine-grained instrumentation for both, application analysis and for application management. Finally, we have provided a comparison to the most important projects in the application analysis and application management domains respectively.

Chapter IV

4 Implementation

4.1 Overview

We have developed our meta-application infrastructure using the Fractal Component Model [Fractal]. Fractal is under constant development and provides all of the functionality required for implementing our infrastructure. Specifically, we have used the Julia [Julia] implementation of the Fractal Model. Julia is the reference implementation for Fractal and is written in Java. We have also used other technologies that are for Fractal and also some technologies that are platform specific to Java. FScript is a script language that provides functionality for introspecting and reconfiguring Fractal applications at runtime. It is useful for inserting dynamic tracers and assigning interaction points for the meta-application. Creating dynamic annotations is particularly time-consuming and has not been implemented. The solution for the Java platform relies on using a bytecode manipulation tool [ASM] and JVMTI [JVMTI], a Java tool for code hot-swapping.

The infrastructure is constructed in three basic parts. The first part is the `Dynamic Tracer Manager`. This part of the infrastructure performs dynamic application instrumentation. Dynamic instrumentation is performed using both dynamic tracers and the asynchronous event annotation toolkit. Tracers are small components that are constructed with two purposes: first, they register inter-component calls that occur in the application, and second, they control the execution flow in the application. Controlling execution provides the necessary time to analyze the events in order to construct the request execution path and also provides an instrumentation point in the application for interposition and meta-application interactions. The annotation toolkit provides information regarding inter-thread communication in order to deduct request execution paths and to perform context operations. The second part of the infrastructure, the `Request Tracker`, constructs the request execution path and performs context related operations. These operations include duplicating contexts, aggregating contexts and creating new context references. The final part of the application, the `Callback Manager` provides the meta-application functionality. It is charged with executing callbacks at interception points. These parts are coordinated using the `Administrator` component. Its purpose is to handle events between different subsections, provide requests to exterior request consumers, and to associate `Tracers` to `Callbacks`.

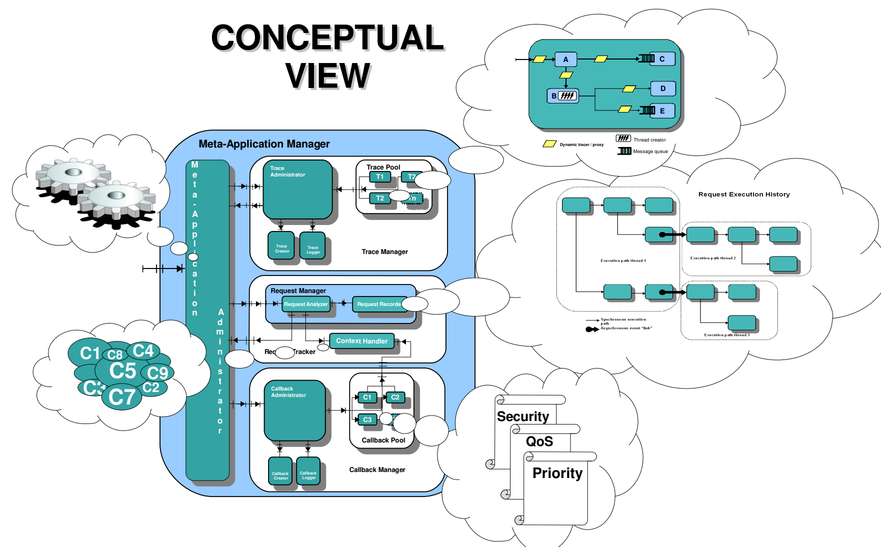

Figure 14: Conceptual view of the implementation.
The figure provides insight on the activities performed by each subsection of the infrastructure. The Trace Manager instruments the application using dynamic tracers. The Request Tracker analyzes events that occur in order to construct request paths and handle contexts. The Callback Manager provides interaction points for meta-application functionality. The Meta-application Administrator directs the whole application, manages tracer – callback relationships, provides external applications with profiling and workload information.

In section 4.2 we present the context in which we have implemented our infrastructure. We have chosen the Fractal Component Model. We explain Fractal and provide an introduction to its reference implementation, Julia. Also, we give an introduction to FScript. In section 4.3 we present the architecture we have designed for the meta-application infrastructure. It is divided into three different architectural elements. They are all coordinated using an `Administrator` component. The first part explains application instrumentation using dynamic tracers and annotations. The second is charged with constructing the request execution path and handling contexts by interpreting the events produced from the instrumentation. The last part explains how we provide interaction points using `Callbacks` for meta-applications constructed using the architecture.

4.2 Implementation Context

4.2.1 Fractal Component Model

Fractal [Fractal] is a modular and extensible component model that can be used with various programming languages to design, implement, deploy and reconfigure various systems and applications, from operating systems to middleware platforms and to graphical user interfaces. Fractal is also a project with several sub projects, dealing with the definition of the model, its implementations, and the implementation of reusable components and tools on top of it.

The Fractal component model heavily uses the separation of concerns design principle. The idea of this principle is to separate into distinct pieces of code or runtime entities the various concerns or aspects of an application: implementing the service provided by the application, but also making the application configurable, secure, available, ... In particular, the Fractal component model uses three specific cases of the separation of concerns principle: namely separation of interface and implementation, component oriented programming, and inversion of control. The first pattern, also

called the bridge pattern, corresponds to the separation of the design and implementation concerns. The second pattern corresponds to the separation of the implementation concern into several composable, smaller concerns, implemented in well separated entities called components. The last pattern corresponds to the separation of the functional and configuration concerns: instead of finding and configuring themselves the components and resources they need, Fractal components are configured and deployed by an external, separated entity.

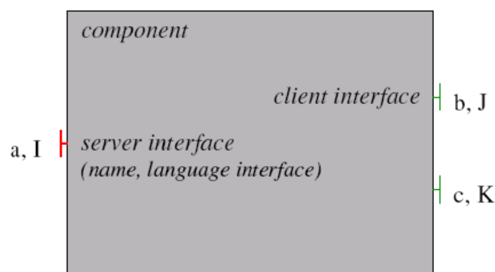

Figure 15: External view of a Fractal component.

The main goals of the Fractal component model are to implement, deploy and manage (i.e. monitor and dynamically reconfigure) complex software systems. These goals motivate the main features of the Fractal model: composite components (to have a uniform view of applications at various abstraction levels), shared components (to model resources), introspection capabilities (to monitor a running system), and configuration and reconfiguration capabilities (to deploy and dynamically reconfigure an application). But another goal of the Fractal model is to be applicable to many software, from embedded software to application servers and information systems. Unfortunately, the advanced features of the Fractal model have a cost that is not always compatible with the limited resources of constrained environments.

4.2.1.1 External component structure

Depending on the level of observation, or *scale*, a Fractal component can be seen as a black box or as a white box. When seen as black box, i.e. when its internal organization is not visible, the only visible details of a Fractal component are some *access points* to this black box, called its *external interfaces* (see Figure 15). In order to invoke operations on a component interface, one must first identify the interface to be called, and then get an access to this interface. In order to access the interface a *binding* must be established to this interface. Each interface has a name, in order to distinguish it from the other interfaces of the component. One may distinguish two kinds of interfaces: a *client* (or *required*) interface emits operation invocations, while a *server* (or *provided*) interface receives them.

4.2.1.2 Internal component structure

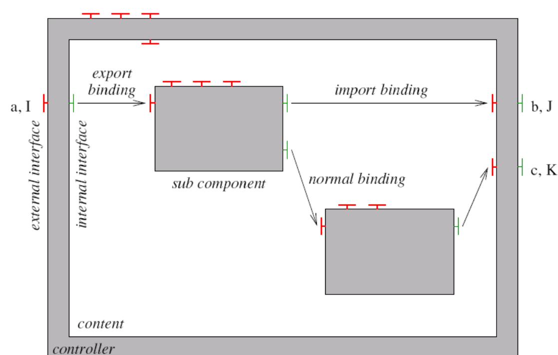

Figure 16: Internal view of a Fractal component.

At the next level of control capability, beyond the "introspection" level where components provide interfaces to introspect their external features, a Fractal component can provide control interfaces to introspect and reconfigure its *internal* features. Internally, a Fractal component is formed out of two parts: a *controller* (also called membrane), and a *content* (see Figure 16). The content of a component is composed of other components, called *sub components*, which are under the control of the controller of the enclosing component. The Fractal model is thus recursive and allows components to be nested at an arbitrary level. A component that exposes its content is called a *composite* component. A

component that does not expose its content, but has at least one control interface is called a *primitive* component. A component without any control interface is called a *base* component.

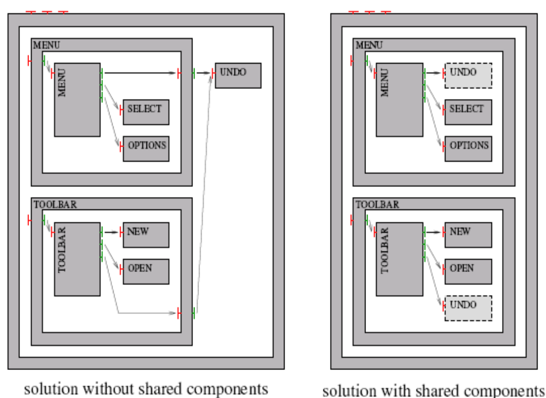

Figure 17: Advantages of shared components

The controller of a component can have *external* and *internal* interfaces. External interfaces are accessible from outside the component, while internal interfaces are accessible only from the component's sub components. A *functional* interface is an interface that corresponds to a provided or required functionality of a component, while a *control* interface is a *server* interface that corresponds to a "non functional aspect", such as introspection, configuration or reconfiguration, and so on.

A component may appear in the content of (be *shared* by) several distinct enclosing components (see Figure 17). A component that is shared among two or more distinct components is subject to the control of their respective controllers. The exact semantics of the resulting configuration (e.g. which control behavior is enacted) is in general determined by an encompassing component that encloses all the relevant components in the configuration.

4.2.1.3 Reconfiguration

Reconfigurations can involve removing a component and replacing it with a new one, adding or removing components. All these operations can be performed dynamically (i.e. while the application is executing). For example, let's say we want to dynamically change a component. In order to do this we need to unbind all of its bindings (i.e. client and server bindings). These unbindings cannot occur unless the component and its parent composite-component are stopped. The new component that will replace this one must be created and added to the same composite component. The old component must be removed. The former bindings that had been undone must be redone with the new component. All stopped components must be restarted. The application can now run with the new component.

4.2.2 Julia, implementation of the Fractal Component Model

Julia [Julia] is the reference implementation of the Fractal component model. Julia is written in Java and is fully compliant with the Fractal Specification. Julia has been designed to be a lightweight and efficient implementation of these specifications. The design choices which have been made aim at reducing the memory footprint and the runtime overhead of Fractal components developed with Julia. Julia is a highly configurable framework which allows creating many different forms of Fractal components. These forms vary depending on the control semantics associated to the component. Julia provides a set of predefined control semantics for frequently used components (e.g. primitive, composite) and allows developers to incorporate their own forms. These forms may redefine or customize any aspect of the control semantics such as lifecycle management, binding creation, naming policies or any other kind of technical service one may want to attach to a Fractal component. Julia uses the ASM [ASM] bytecode engineering library for constructing at runtime a Fractal component instance. ASM is used in many different situations:

- to generate interceptor and Fractal interface instances,

- to perform optimizations such as merge strategies for reducing memory footprint
- to modularize the writing of control classes with a mixin algorithm which generates the bytecode of a class from several different layers developed independently.

Since version 2.5, Julia provides the notion of a component-based control membrane. The idea is to define the control semantics of a Fractal component with the assembling of other so-called control components. These control component are themselves Fractal compliant (they implement the Fractal API) and their assembling is described with Fractal ADL. This feature is described in Section 9 of the Julia API documentation.

4.2.3 FScript for Safe Dynamic Reconfigurations

The Fractal APIs provide dynamic discovery and reconfiguration operations, however, with certain drawbacks. The Fractal APIs are minimalist and orthogonal, causing code to be verbose and not very readable. Since Fractal introduces new concepts that are not implemented in the host language (e.g. components, bindings), and other concepts that are used differently (e.g. interfaces), developers may be confused. Furthermore, in the case of Fractal implemented in Java, Java is a general purpose language and does not provide guarantees when executing Fractal reconfiguration code. Such guarantees could be insuring that data structures are not corrupted, calling dangerous methods, or simply looping forever. To overcome these limitations and retain Fractal's advantages, a new Domain Specific Language has been implemented, namely FScript [David, 2006]. The language is used for navigating inside Fractal architectures and dynamically reconfiguring them. FScript uses a special notation called FPath [David, 2006] to navigate intuitively inside an architecture and select parts of it. FScript has been implemented as a simple interpreter, that can be embedded inside Fractal applications.

FPath is a special notation used inside the FScript language to *navigate* inside Fractal architectures and *select* elements in it according to some predicate. Its syntax and execution model are inspired by the XPath language which solves the same problem on XML documents (although FPath does not use XML). FPath sees a given Fractal architecture as an oriented graph with labeled arcs. Different kinds of nodes represent all the architectural elements reified: the *components* themselves, component *interfaces* (both external and internal), configuration *attributes* corresponding to getter/setter methods, and finally *methods* on the interfaces. These nodes are connected by labeled arcs, which denote the kind of relation between them. The following types of arcs, called axes are defined in FPath:

- `component`: from any kind of node to the component owning this node;
- `attribute`: from a component node to all its configuration attributes;
- `interface`: from a component node to all its interfaces, and from a method node to the interface of which it is part;
- `method`: from an interface to all its methods;
- `binding`: from an client interface node to the server interface it is bound to, if any;
- `child` (resp. `parent`): from a component to its direct children (resp. parents);
- `sibling`: from a component to all the other components which have at least one direct super-component in common with it;
- `descendant` (resp. `ancestor`): from a component to all its direct and indirect children (resp. parents). `descendant` (resp. `ancestor`) is thus the transitive closure of `child` (resp. `parent`).

FPath expressions denote *relative paths* starting from an initial (set of) node(s) in the graph. Such a path is made of a series of steps, each made of up to three elements: `axis::test[predicate]` (the predicate is optional). On each step, an initial set of nodes is converted to a new set by following all the arcs with a label corresponding to the axis, then filtering the result using the *test* (on the node names) and optional *predicates* (boolean expressions applied to each candidate). For a multi-step path, this algorithm is repeated with the result of the previous step as the current node-set of the next.

For example, `sibling::*/interface::*[provided(.)][not(bound(.))]` is made of two steps. The first one uses the `sibling` axis, an "empty" test `*` (which is always true) and has no predicate. The second step uses the `interface` axis, no test either, and two predicates which are combined. Inside the predicates, the dot "`.`" represents the current node on which the predicate is evaluated. Evaluating the complete expression starting from an initial component node will:

1. select all its sibling components, however they are named;
2. select all the external interfaces of these siblings;
3. filter this set of interfaces to return only server interfaces (`provided()`) which are not already bound.

4.2.4 FScript Reconfigurations

The preceding section described the FPath notation which is used to navigate inside a Fractal architecture and select parts of it, but cannot modify the architecture. The complete FScript language, of which FPath is just a part, enables the definition of *reconfiguration actions* to apply to a running application. FScript is a simple imperative/procedural language whose main features are:

- direct syntaxic support for navigation in Fractal architectures thanks to FPath;
- safety guarantees on the application of the reconfigurations;
- a very dynamic implementation which does not impose a compilation phase and can be easily embedded into existing applications, where reconfiguration scripts can then be dynamically loaded and executed.

FScript distinguishes two kinds of procedures: functions and actions. Functions are guaranteed to be side-effect free, and can only introspect an architecture, not modify it. They can be used safely inside FPath requests, for example in the predicates. Functions are defined like actions, expect that they use the `function` keyword instead of `action`, and can only invoke other functions, not actions (be they primitive or user defined). FScript provides a standard library of primitive functions and actions which gives the user access to all the information available from the Fractal API, and all the standard reconfigurations.

FScript's design and implementation guarantee the consistency of reconfigurations. Because these reconfigurations are applied to running applications, it must be guaranteed that they will not break the target system. To this end, they have chosen a set of consistency criterion, in particular *transactional integrity* (atomicity, consistency of the final state, isolation) and *termination* of the reconfigurations. The validation of these criteria is guaranteed in part by the language's structure itself, whose expressive power has been limited, and in part by the implementation.

4.3 Application management infrastructure

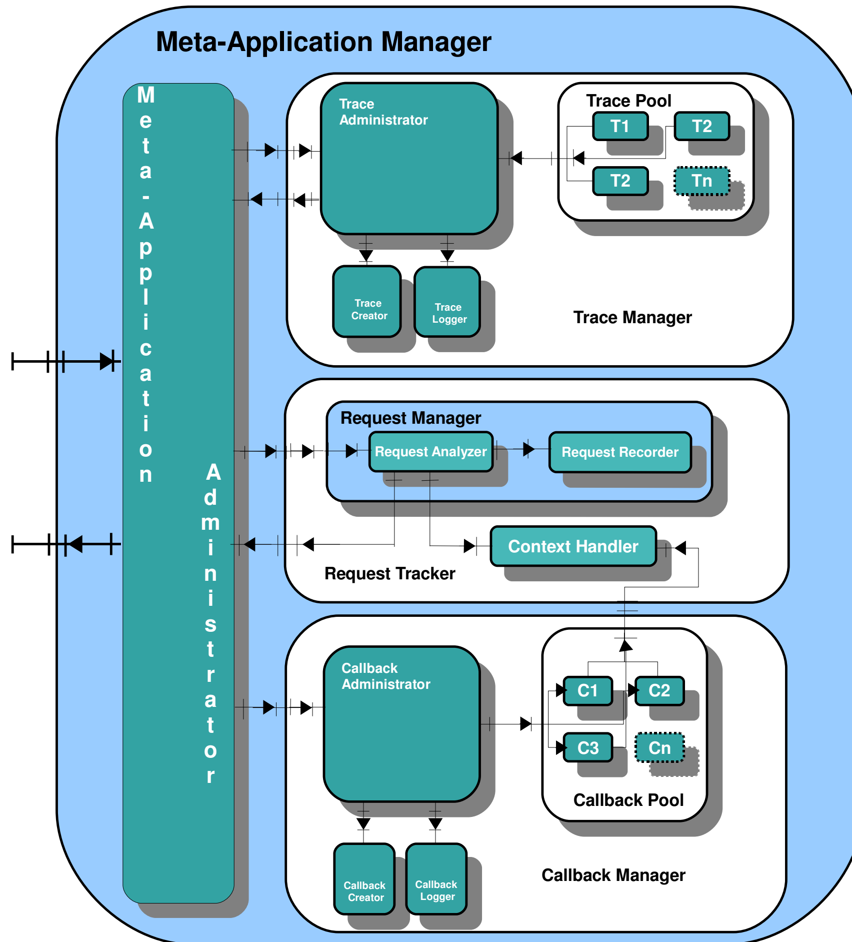

Figure 18: Meta-application infrastructure

The infrastructure shows the separation of the phases required for implementing the meta-application infrastructure. It is composed of the Trace manager, the Request Tracker and the Callback Manager. All events pass through the Meta-application Administrator, which performs associations between tracers and callbacks, filters unwanted events, and passes events though the system.

We present our meta-application infrastructure (see Figure 18). It has been created using the Fractal component model. It instruments the application and provides meta-application and profiling functionality for Fractal component applications. The architecture is divided into three basic parts. The first part is the `Trace Manager`, which creates and inserts dynamic tracers into applications. The second part is the `Request Tracker`, that analyzes events provided from application instrumentation. The third part is the `Callback Manager`, which implements the meta-application functionality specified by the user. These three parts will be presented in detail.

4.3.1 Trace Manager

The `Trace Manager` performs dynamic application instrumentation. The application is instrumented using dynamic tracers and the asynchronous event annotation toolkit. The `Trace Manager` handles only dynamic tracers. It is composed of several components, a `Trace Administrator`, a `Trace Logger`, a `Trace Creator` and a `Trace Pool`. The `Trace Administrator` correlates the different components and instruments the application. The `Trace Manager` receives a reference to the component to be instrumented. Using FScript, the `Trace Manager` performs a search for all bindings in the application. For every binding in the application a unique and compatible tracer must be created. The `Trace Creator` is used for this purpose. Each tracer must implement a server and a client interface compatible with the binding to be instrumented, in addition to a synchronous event logging interface used for notifying the meta-application that a call is being performed. Since Fractal applications can be modified dynamically, there is no way of knowing what interfaces need to be implemented before runtime. Creating dynamic components with unknown interfaces can be performed in two different ways: using a bytecode editor [ASM] to create the component, or using Java Reflection and specifically Dynamic Proxies to imitate a components behavior. We have chosen the second solution because it is a quicker solution to implement, although Java Reflection generates more overhead and there are more limitations than the bytecode editor solution. Tracers are a generic component that is partially created before runtime, and partially created using the Dynamic Proxy API [JavaProxy]. More specifically, a generic tracer exists before execution time, and it is wrapped in a component wrapper with the necessary client and server interfaces implemented. After the tracer is created, the components involved are temporarily halted, the binding in the application is unbound, the tracer component is added to the composite component, the client interface in the application is bound to the dynamic server interface of the tracer, and the client interface of the tracer is bound to the server interface of the application. All calls through that binding are now intercepted by the tracer. All bindings in the application are instrumented in the same fashion and all inter-component activity is now visible to the meta-application infrastructure.

When a call is intercepted by a dynamic tracer, the tracer notifies the `Trace Administrator`. The `Trace Administrator` uses the `Trace Logger` for recording the event so it can be later analyzed if necessary. Then the `Trace Administrator` notifies the `Meta-application Administrator` of the event. Specifically, the information sent provides details about the tracer that has intercepted the call, the thread ID of the halted thread, and the event that has occurred (i.e., *pre*, *post* or *error*). The `Trace Administrator` is then in charge of the event and will be explained later.

4.3.2 Request Tracker

The `Request Tracker` receives and analyzes synchronous and asynchronous execution events in the application, records the execution path of the requests, and also performs context related operations. The `Request Manager` handles the analysis of the event and indicates what to do to the `Request Recorder` and the `Context Handler`. In the case of synchronous events, the handler analyzes the thread ID in question and the nature of the event. If it is a call (i.e. *pre*), the `Request Recorder` is notified to add a new component to the thread execution path in question. If it is a return (i.e. *post*) of an open call, the `Request Recorder` is notified and the call is closed. If all calls of the thread execution path are closed, then the thread has finished servicing that request. If all threads have finished servicing the request and no messages or other entities related to the request are waiting for service, then the request has finished. If the event is an error event, then the call is closed and the same analysis as before regarding termination of the request is

performed. Because threads are entities servicing a request and the thread execution path is already related to the request execution path, adding calls to a thread execution path implies adding it to the request execution path itself.

Asynchronous events are treated differently because asynchronous links must be created between thread execution paths and the Context Handler must modify contexts according to the event. If a thread is created or called from a pool, then the called thread is now dependent of the caller thread. This forks the request execution path. The `Request Recorder` is notified and the open call from the caller thread execution path has an asynchronous link added to it. This asynchronous link records the details of the event, including the time the event occurs, and it points to the thread execution path of the dependent thread. The two threads continue execution and their events continue to be analyzed. The `Context Handler` must be notified when this occurs because it associates the called thread to the `request contexts` of the caller thread, and it duplicates the `message context` of the caller thread and associates the duplicated version of the `message context` to the called thread. If the event is a `message sent`, then the open call of the thread execution path performing the write is added an asynchronous event link. The link is left open, and will be closed when the message is read or received. The `Context Handler` is notified and the `request contexts` of the writing thread are associated to the message, and the `message context` of the writing thread is duplicated and associated to the message. If the event is a *message received*, the *Request Recorder* is notified and the open asynchronous link is closed, with a link to a newly created thread execution path representing the execution path of the thread that has read the message. The thread is now associated to the *request contexts* of the message and the *message context* of the message. If the thread already had contexts associated to itself, then the contexts are added (see 3.5.5 Handling multiple contexts). The message and the associated contexts are removed from the *Context Handler* and the *Request Recorder*. If the event was a *message read*, the same actions as for message received are performed, except the message and the contexts are not removed from the *Context Handler* or the *Request Recorder* because the message can be read by another thread at a later time.

When requests finish, they notify an application using the *Request Consumer Interface*. They send the full request execution history, including request statistics (e.g., per component latencies, request latency, amount of thread execution paths), to the consumer of the event. This provides a means for performing workload analysis and application profiling.

4.3.3 Callback Manager

The `Callback Manager` is used for implementing the meta-application functionality described by the user. For a `Callback` to be used two elements are required, the interception points in the application and the callback functionality. The interception points are controlled by the `Trace Administrator.` The `Callback Manager` has no direct knowledge of them. The `Callback Creator` creates the `Callback` component by receiving a Java object that implements the `Callback Interface`. The `Callback Interface` consists of three methods, a `pre,` a `post` and an `error`. These methods correspond to the events that a tracer produces. The `Callback` component is created using the functionality described in the Java object provided by the user. It is then added to the `Callback Pool` and assigned an ID. This ID is returned when the component is created and is later used by the `Meta-application Administrator` to associate `Callbacks` to `Tracers`. These associations are hidden from the `Callback Manager`.

When a tracer notifies that a synchronous communication event has occurred, the `Meta-application Administrator` decides which `Callbacks` need to be executed. It send the IDs of the `Callbacks`. The ID of the `Callback` is used to identify it. The `Callback`

`Administrator` receives the petition, it logs the petition using the `Callback Logger`, and it executes the required `Callback`. The `Callback` can access `request contexts` and the `message context` associated to the thread that was halted by the dynamic tracer. It cannot access other contexts in the application. A `Context Access Interface` is required by each `Callback` and is provided by the `Context Handler`. A `Callback` performs the tasks assigned by the user and has full access to context modification or consultation. When finished, the call is closed and control returns. The tracer then permits the thread to continue execution.

4.3.4 Meta-application Administrator

The `Meta-application Administrator` proxies calls between the different sub-components of the meta-application, and between sub-components and the exterior. It is used to isolate cross-component functionality, like interception point – callback associations. This helps make the the `Trace Manager`, `Request Tracker` and `Callback Manager` less dependent on each other.

The `Meta-application Administrator` receives the request to instrument an application. It notifies the `Trace Manager`, which then inserts dynamic tracers in the application. For full application instrumentation the asynchronous event annotations must also be inserted. At the moment the annotations are not dynamic, and the application is compiled with them. A future edition of the infrastructure may include an `Annotation Manager` that, using a bytecode manipulation toolkit [ASM] it records the placement of the annotations and removes the annotations from the application either at compile-time or at load-time. When instrumenting the application, the annotations may be reinserted into the application at runtime using Java code hot-swapping technologies [JVMTI]. This solution is feasible but requires time to implement. At the moment the `Meta-application Administrator` receives the asynchronous events directly. The `Administrator` also receives requests for meta-application creation. For each `Callback` component in the application, an FPath query and a Java object containing the code for the meta-application at the specific points specified by the FPath query. The Java object is sent to the `Callback Manager` and the `Callback` is created. The FPath query may contain a single interface, or a set of interfaces to be instrumented. The `Meta-application` administrator performs an analysis of the interfaces and locates the dynamic tracer already instrumenting each interface. For each dynamic tracer an association with the `Callback` specified by the user is made. One `callback` can be associated to many different dynamic tracers (see 3.6 Callback infrastructure). After the application is fully instrumented and the meta-application behavior instantiated, the meta-application infrastructure commences receiving calls (events may occur before all instrumentation is inserted or before all callbacks are created). A dynamic tracer is activated and sends an event to the `Meta-application Administrator` when a call between two components occurs. The `Administrator` notifies the `Request Tracker` and then reads the list of `Callbacks` associated to the tracer and sends the ordered list of Callback IDs to the `Callback Manager` for execution. When asynchronous events occur, the `Administrator` is notified directly. It passes the event to the `Request Tracker` who performs the necessary tasks. When a request is finished, the `Request Tracker` notifies the `Administrator`. The `Administrator` contains a list of request consumers that are interested in finished request information. The request is duplicated and an individual request execution path is sent to each consumer through the `Request Consumer Interface` (messages are duplicated because components cannot share them).

4.3.5 Current state and remaining work

In this chapter we have presented our architecture for performing application management. Currently, we have implemented the asynchronous event annotation toolkit, dynamic tracers, request tracking and callbacks. The annotation toolkits are not currently dynamic. They have been implemented as static method calls that are directly inserted into the application's code. We have implemented request tracking and context propagation, and they correctly interpret synchronous and asynchronous events. We have also implemented the callbacks. At the moment, we are in the phase of evaluating the architecture and its feasibility. For these evaluations we require a component application that can be easily analyzed without our architecture, so we can compare the results with our expectations. We plan on using the Comanche Web-Server [Comanche], and extending its functionality, adding asynchronous events and creating different execution paths in the application. Once Comanche is modified, we can test the overhead from dynamic tracers, the annotation toolkit, context propagation, request tracking and callbacks (although callback overhead depends on user specified functionality). We also plan on constructing a profiling and workload analysis to prove the feasibility and usability of the `request consumer interface`. If successful, the interface would provide means for performing online application analysis based on common instrumentation, thus unifying application management and analysis. To conclude the initial evaluations of our project, we must provide a method of focusing on points of interest in the application and discriminating annotations. In case overhead from the application is too high, discrimination of events could provide a means of overcoming this limit.

Chapter V

5 Conclusion

Summary of the Contribution

Modern applications are more and more complex. Tools for understanding applications are essential for management and analysis tasks. Currently these tools are lacking and have not kept pace with tools for developing the applications. Performance problems are hard to diagnose and constructing accurate models of a system's workload is difficult. Furthermore, software components intertwine their functional concerns (e.g., their specific job) with their non-functional concerns (e.g., priority, QoS, security), limiting the reusability of the software components themselves. We propose unifying *application analysis* and *application management* to provide a coherent view of what currently are disjoint concepts. A solution that unifies application analysis and application management must be fine-grained, dynamic and produce low-overhead. We have provided such a unification by means of a generic fine-grained *instrumentation infrastructure*. We have defined a common base-granularity that is beneficial for both, application management and application analysis. We propose *requests* as the granularity for analysis and management.

Based on the request-granularity, we have presented the necessary techniques and information for *automated request tracking*, *context propagation* and *meta-application* functionality. We present dynamic instrumentation in component-based applications by means of dynamic tracers. We give a thorough analysis of asynchronous communication events that exist in component applications and how to follow causal information paths through the application. We have paid special attention to correctly interpreting causal pathways in the application, and to providing dynamic instrumentation and interception points. We also have proposed two novel forms of contexts, namely *message contexts* and *request contexts*. The solution provides unified fine-grained instrumentation for both, application analysis and for application management.

Our solution is applicable to all component models that support introspection and dynamic reconfigurations. Our meta-application infrastructure provides fine-grain interaction points on every component interface, it is run-time dynamic, and it produces zero overhead when not enabled. Complementary to existing techniques in application analysis, like performance analysis or application profiling, we provide a request consumer interface, that can easily feed request execution paths into external application analysis tools. Our application management infrastructure has been implemented using the Fractal Component Model. Specifically, we have used the reference implementation of Fractal, Julia.

Future work

Due to lack of time, the quantitative evaluation (i.e., overhead measurements) remains to be done on realistic applications. We intend to fully quantify and minimize the overhead produced by our application. If overhead is low enough the solution should be feasible for production environments. We plan on extending the functionality of our infrastructure to interact with multiple component applications simultaneously, and on analyzing how to propagate contexts across multiple applications and across networks. Some existing solutions provide grouping metadata into TCP/IP packets for network transmission, but this limits the amount of metadata that can be sent.

We also plan to use a domain specific language (DSL) to improve user interaction with the application. This language must be capable of controlling dynamic tracer insertion and the granularity of the analysis, providing analysis in specific points of interest. It must select which annotations are to be analyzed and which dynamic tracers to be deployed, thus avoiding the cost and overhead of analyzing non desirable annotations and portions of the application. Also, the language must be capable of specifying callback interaction points and of defining callback functionality, fully extracting the meta-application behavior from the application under study. Extra benefits to the language could be guaranteeing safe execution, by using a bytecode interpreter.

Bibliography

[Aguilera et al.,2003] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03), 2003

[ASM] http://asm.objectweb.org/

[Barham et al., 2003] P. Barham, R. Isaacs, R. Mortier, and D. Narayanan. Magpie: online modelling and performance-aware systems. 9th Workshop on Hot Topics in Operating Systems (HotOS IX), 2003

[Bellard] F. Bellard. Qemu: Open source processor emulator. http://fabrice.bellard.free.fr/qemu/about.html, 2007

[Bershad et al., 1995] Brian Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gun Sirer, David Becker, Marc Fiuczynski, Craig Chambers, and Susan Eggers. Extensibility, safety and performance in the SPIN operating system. In Proceedings of the 15th ACM Symposium on Operating System Principles, 1995

[Cantrill et al., 2004] Bryan M. Cantrill, Michael W. Shapiro and Adam H. Leventhal. DTrace: Dynamic Instrumentation of Production Systems. USENIX Annual Technical Conference, 2004

[Chanda et al., 2007] Anupam Chanda, Alan Cox and Willy Zwaenepoel. Whodunit: transactional profiling for multi-tier applications. Eurosys, 2007

[Chanda et. al, 2005] Anupam Chanda, Khaled Elmeleegy, and Alan L. Cox. Causeway: Operating System Support For Controlling And Analyzing The Execution Of Distributed Programs. Proceedings of the ACM/IFIP/USENIX 6th International Middleware Conference, 2005

[Chen et al., 2002] M. Chen, E. Kiciman, E. Fratkin, E. Brewer, and A. Fox. Pinpoint: Problem determination in large, dynamic, Internet services. International Conference on Dependable Systems and Networks, 2002

[Comanche] http://fractal.objectweb.org/tutorial/index.html

[David, 2006] Pierre-Charles David. Safe Dynamic Reconfigurations of Fractal Architectures with FScript. The 5th Fractal Workshop at ECOOP, 2006

[Eclipse] http://www.eclipse.org/

[Fractal] http://fractal.objectweb.org/

[GDB] http://sourceware.org/gdb/

[Gunter, 2005] Daniel K. Gunter and Brian L. Tierney. Scalable Analysis of Distributed Workflow Traces. International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'05), 2005

[Hastings et al., 1992] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. Proceedings of the Winter USENIX Conference, 1992

[Hrishchuk et al., 1995] Curtis E. Hrischuk and Jerome A. Rolia and C. Murray Woodside. Automatic Generation of a Software Performance Model Using an Object-Oriented Prototype. In Proc. MASCOTS, 1995

[Isaacs et al., 2005] Rebecca Isaacs, Paul Barham, James Bulpin, Richard Mortier, and Dushyanth Narayanan. Request extraction in Magpie: events, schemas and temporal joins. 11th ACM SIGOPS European Workshop, 2005

[JavaProxy] http://java.sun.com/j2se/1.5.0/docs/api/java/lang/reflect/Proxy.html

[Julia] http://fractal.objectweb.org/julia/

[JVMTI] http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/jvmti.html

[Kiczales et al., 2001] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kirsten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In Proceedings of the 15th European Conference on Object-Oriented Programming, 2001

[Log4j] http://logging.apache.org/log4j/docs/index.html

[Miller, 1984] Barton P. Miller. A Distributed Programs Monitor for Berkeley UNIX. University of California at Berkeley, 1984

[Moore, 2001] Richard J. Moore. DProbes: A universal dynamic trace for Linux and other operating systems. Proceedings of the FREENIX Track, USENIX Tech, 2001

[Olszewski et al., 2007] Marek Olszewski, Keir Mierle, Adam Czajkowski, Angela Demke Brown. JIT instrumentation: a novel approach to dynamically instrument operating systems. Eurosys, 2007

[Prasad et al., 2005] Vara Prasad, William Cohen, Frank Ch. Eigler, Martin Hunt, Jim Keniston and Brad Chen. Locating System Problems Using Dynamic Instrumentation. Linux Symposium, 2005

[Qie, 2002] Xiaohu Qie, Ruoming Pang and Larry Peterson. Defensive programming: using an annotation toolkit to build DoS-resistant software. OSDi, 2002

[Reumann et al., 2004] J. Reumann and K. G. Shin. Stateful Distributed Interposition. ACM Transactions on Computer Systems, 2004

[Schiavoni, 2006] Valerio Schiavoni and Vivien Quema. A posteriori defensive programming: an annotation toolkit for DoS-resistant component-based architectures. Proceedings of the 2006 ACM symposium on Applied computing, 2006

[Seltzer et al., 1996] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. VINO: Dealing with disaster, surviving misbehaved kernel extensions. Proceedings of the Second Symposium on Operating Systems Design and Implementation, 1996

[Srivastava et al., 1994] Amitabh Srivastava and Alan Eustace. ATOM: A system for building customized program analysis tools. Proceedings of the ACM Symposium on Programming Languages Design and Implementation, 1994

[Tamches et al., 1999] Ariel Tamches and Barton P. Miller. Kerninst: fine-grained dynamic instrumentation of commodity operating system kernels. Proceedings of the Third Symposium on Operating Systems Design and Implementation, 1999

[Wisniewski et al., 2003] Robert W. Wisniewski and Bryan Rosenburg. K42: efficient, unified, and scalable performance monitoring for multiprocessor operating systems. In SC'2003 Conference CD, 2003

[Yaghmour et al., 2000] Karim Yaghmour and Michel R. Dagenais. Linux Trace Toolkit (LTT): Measuring and characterizing system behavior using kernel-level event logging. Proceedings of the 2000 USENIX Annual Technical Conference, 2000