

# Request extraction in Magpie: events, schemas and temporal joins

Rebecca Isaacs, Paul Barham, James Bulpin\*, Richard Mortier, and Dushyanth Narayanan  
Microsoft Research, Cambridge, UK.

## Abstract

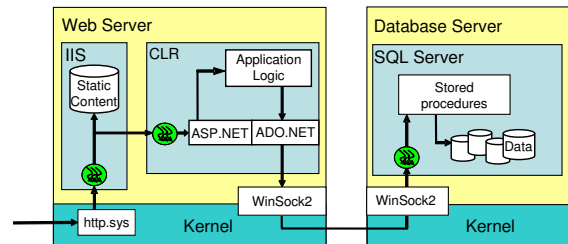
This paper addresses the problem of extracting individual request activity from interleaved event traces. We present a new technique for event correlation which applies a form of **temporal join** over timestamped, parameterized event streams in order to identify the events pertaining to an individual request. **Event schemas** ensure that the request extraction mechanism applies to any server application or service without modification, and is robust against future changes in application behavior. This work is part of the Magpie project [2], which is developing infrastructure to track requests end-to-end in a distributed system.

## 1 Introduction

Detection and diagnosis of faults in server systems is still something of a black art. Transient performance problems or functionality glitches are the norm for users of multi-tier web sites. Although straightforward programming errors and hardware failures are likely to be the root cause of most problems, the effects are exacerbated and the causes obscured by the interactions of multiple machines and heterogeneous software components. The traditional approach to monitoring system health is to log aggregate performance counters and raise alarms when certain thresholds are exceeded. This is effective for identifying some problems, but will not catch others such as poor response time or incorrect behaviour (“why was the item not added to my shopping cart?”). The latter class of problems is specific to individual user requests, and can often only be identified by tracking the requests through the system, monitoring each one as it traverses components and machines.

We believe that request tracking is an important aspect of the larger challenge of understanding complex system behavior. Fault detection and diagnosis is already feasible using request path analysis [4]. Magpie constructs models which characterize the workload for performance prediction and performance debugging [2]. A

\*University of Cambridge, work done while at Microsoft Research



**Figure 1.** Application and operating system components involved in processing an HTTP request to a two-tier web site. Thread pools are shown as small, shaded circles.

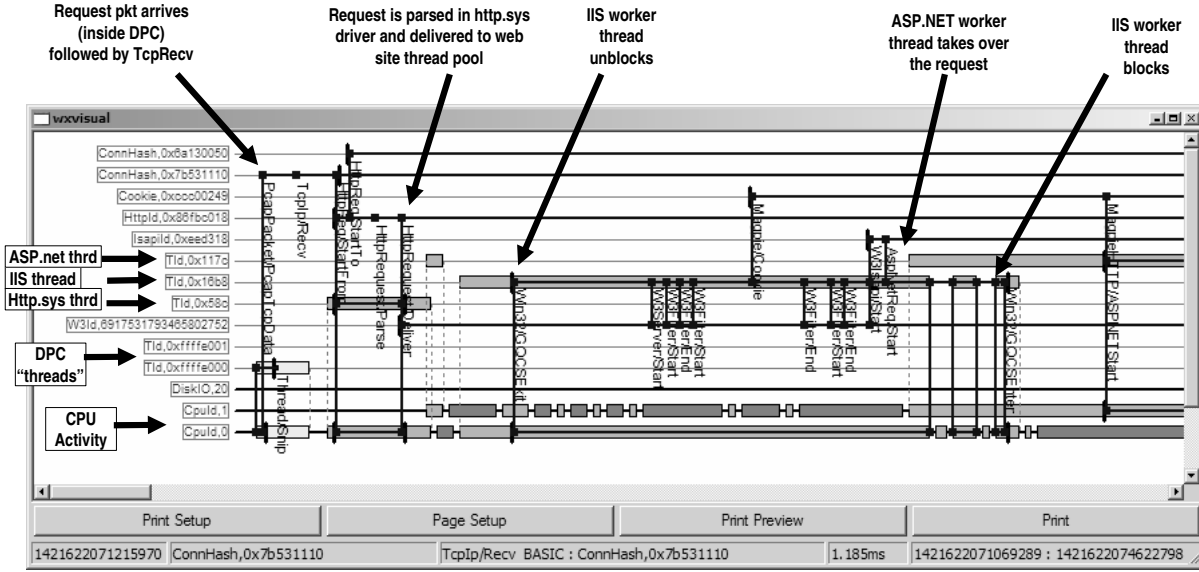
compelling vision for the future is of systems that are performance-aware and self-configuring, tuning themselves automatically in response to changes in the workload [6].

The first requirement for request tracking is a general purpose mechanism for extracting individual request activity from interleaved event traces. It is important that any such tool, just like the system instrumentation, be re-usable for many applications. We have therefore developed a request parsing algorithm which relies on an *event schema* to describe the semantics of events for the particular application of interest, and *temporal joins* to combine related events into requests.

In the following section we clarify the term request, before describing how the event schema is used in conjunction with temporal join to give an elegant, general-purpose and efficient mechanism for request extraction.

## 2 Requests and events

A **request** is system-wide activity which takes place in response to any external stimulus of the application being traced. For example, the stimulus of an HTTP request may trigger the opening of a file locally or the execution of multiple database queries on one or more remote machines. In other application scenarios, database



**Figure 2.** Screenshot from request parser visualization tool with the request sub-graph highlighted. Each attribute-value pair is shown as a horizontal line with time increasing from left to right. When a thread is active on a CPU its timeline is overlaid with a coloured rectangle. The vertical dotted lines represent context switch events, elided for clarity. Events are shown as vertical bars with square boxes binding them to one or more attribute-value pairs, and are labelled with their names. This screenshot shows a small excerpt of a single HTTP request.

queries may be considered requests in their own right. Within Magpie both the functional progress of the request and its resource consumption at every stage are recorded. Thus a request is described by its path taken through the system components (which may involve parallel branches) together with its usage of CPU, disk accesses and network bandwidth.

An **event** consists of a machine-local timestamp, an identifier and zero or more named attributes. For example, a processor context switch event,

$$cswitch(time=t, cpu=0, tid-in=T_1, tid-out=T_0)$$

records that at time  $t$ , thread  $T_1$  was swapped in on CPU 0 and thread  $T_0$  was swapped out.

Throughout this paper we use as a running example the initial processing of an HTTP request on a two-tier web service consisting of Microsoft’s IIS web server and SQL Server database, both running over Windows. Figure 1 shows the major application and operating system components. IIS serves static content using an in-kernel driver with its own cache and has a thread pool for each web site. Active content typically causes execution of managed code within the .NET Common Language Runtime (CLR), synthesizing HTML using ASP.NET, or accessing a database via ADO.NET.

Even in this simple, yet not atypical example site, the

heterogeneity of components and the complexity of their interactions poses challenges for any request tracing mechanism. ASP.NET operates another (managed) thread pool, and ADO.NET uses WinSock2 to make RPCs to the database. SQL Server employs a heavily-optimized scheduling mechanism with its own thread pool abstraction.

Figure 2 contains a screenshot from our visualization tool showing how the events pertaining to the example request are correlated by the request parser. Notice that there are three threads working on this portion of the request: the Http.sys thread, an IIS worker thread and an ASP.NET worker thread. Control flow is tracked from one thread pool to another by means of event joins, as we now explain.

### 3 Temporal joins

Different event types frequently refer to the same attribute, and this allows us to *join* events on the value of that attribute. Using transitivity, we grow sets of related events which eventually form entire requests.

In the example request of Figure 2, the `HttpRequest/Deliver` event posted by Thread 0x58c has a `W3Id` attribute. A short time later, thread

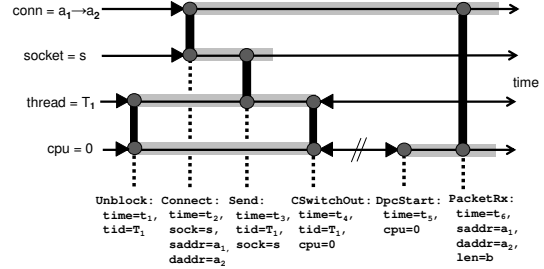
Referring again to Figure 2, the period between the IIS worker thread unblocking (indicated by the Win32/GQCSExit event) and blocking (indicated by the Win32/GQCSEnter event) forms a valid-interval for *ThreadId*=0x16b8. During this period every event posted with a *ThreadId* attribute of 0x16b8 will belong to the same HTTP request.

The transitive joining of events as described above not only enables the control path of a request to be tracked, but also its resource consumption. Certain events are associated with the use of a physical resource, obviously packets and disk read or write events indicate usage of network and disk respectively, while context switches and interrupts allow accounting of CPU cycles to requests. Often attributing these resource consumption events can only be achieved by the transitive join of several events.

Figure 3 illustrates the use of transitive joins for resource usage attribution by showing how a packet event is joined to a send event via an earlier socket connect event. This associates the packet of size  $b$  bytes to the request which thread  $T_1$  was working on at the time of the send event, even though the packet event itself has no explicit reference to a particular request and  $T_1$  has been swapped out at the time the packet is transmitted. Similarly, joining the packet event to the request transitively joins the `DpcStart` event<sup>1</sup>, and hence accounts the CPU cycles consumed in the network stack to the same request.

The join attributes and the events which delimit their valid-intervals are expressed in an application-specific *schema*. A request is made up of a set of related events and is extracted by repeatedly executing temporal joins against an event stream according to schema rules. We refer to this process as request *parsing*, and describe the design and implementation of the Magpie parser next.

<sup>1</sup> A Deferred Procedure Call (DPC) is a form of software interrupt used in the NT kernel. DPCs are used to run device driver I/O completion routines, e.g. to perform network protocol receive processing.



**Figure 3.** Transitive joins enable packet bandwidth and CPU consumption by the network stack to be correctly attributed, even though the thread which issues the send request is swapped out at the time the packet is transmitted. Note that this diagram follows the conventions of the auto-generated visualizations of real requests, with each attribute-value pair shown as a horizontal line and events represented as connecting vertical lines.

## 4 Magpie request parser

Our approach to request extraction is strongly influenced by related work from the field of temporal databases [8]. In such databases, every row in a table has a valid-interval and queries can be issued against the data as of *any* specified time. The main drawbacks to using a temporal database for Magpie request extraction are that off-the-shelf products are not available, and fully general temporal representations and transactional semantics are unnecessarily heavyweight in this context. Our problem is significantly simpler than the general case.

The request parser looks at each event from the event stream in time order and speculatively build up sets of related events. Some of these sets will eventually be identified as describing a complete request, others can be discarded. Because the way in which events are related is defined out-of-band in an application-specific schema, the request parser itself contains no builtin assumptions about application or system behavior.

Within the schema, a valid-interval is specified by associating *binding* types with event types. When an event indicates the start of a valid-interval for a particular attribute-value pair, then we specify its binding type as **START**. To indicate the closure of the valid-interval, the event is given a **STOP** binding. All other events have a **BASIC** binding. This is visualized in Figure 2, where each event is shown as a vertical line with a small square on the appropriate attribute-value timeline. **START** bindings have a vertical line to the left of the square and **STOP** bindings have a line to the right. A valid-interval can be

seen in the figure delimited by the unblocking and blocking events of thread 0x16b8, which is excluded from the request to the left of the Win32/GQCSExit event and to the right of the Win32/GQCSEnter event.

A request is formed in the parser by taking the set of events which are reachable from some seed event by the transitive closure of join operations. The seed event will be an event that occurs only and always within a request. In the case of our running example, it is the HTTP request start event.

## 4.1 Implementation

In the Magpie prototype instrumentation is provided by Event Tracing for Windows (ETW) [7], a highly efficient logging infrastructure which allows us to easily attach event providers to components as required. Consequently the request parser is implemented as an ETW consumer, which takes a schema and an event stream and produces representations of individual requests. It can function either online or offline and there are trade-offs with each approach. In the former it is not required to write events to persistent storage but only one request schema can be applied at a time. In the latter, it is possible to parse the same set of events multiple times, using different request schemas.

An ETW event consumer iterates through buffers of events and issues a callback for each event in timestamp order. The parser inspects the event parameters and for each one creates a binding of the attribute to the new value. Current values of each attribute are maintained in a hash table which maps from each (*Attribute*, *Value*) pair to an event *timeline* - a time-ordered list of all events which make the same particular binding. Event timelines record valid-intervals by use of START, STOP and BASIC bindings. A valid-interval is therefore just a contiguous section of the event timeline bounded by START and STOP events.

Temporal joins are implemented by binding events to one or more timelines. For example, the web server schema used to parse the request shown in Figure 2 states that a W3ServerStart event joins the event's *HttpId* and *ThreadId*. Thus when the parser sees a HttpRequestStartFrom event with fields *HttpId*=0x88fbc018 and *ThreadId*=0x58c, it enters this single event onto the list containing all events with *HttpId*=0x88fbc018 and onto the list containing all events with *ThreadId*=0x58c, creating new lists as required. Since this single event appears in the two lists, these attributes are now joined.

As events are processed one by one, sub-graphs of re-

lated events are constructed. From time to time, some of these sub-graphs will become unreachable because all valid-intervals have been closed. If the sub-graph contains the necessary request seed event, then the request can be output, otherwise it will be garbage collected. Note that there will be many lists (or graphs) that turn out not to represent any request, but instead contain events that are generated by other applications and background system activities. We employ an additional timeout to bound the resources consumed by such activities.

Our first implementation comprises approximately 9000 lines of C code. It takes 7 seconds on a 2.4GHz P4 to parse logs from a 2 minute TPC-C stress test containing 558 requests, where each request on average consists of 411 events. The space requirements are bounded by the maximum duration of a request, with the absolute volume of events determined by the rate of event generation on the system. The logs for this run contained approximately 1.08 million events—our very fine granularity logging includes every context switch.

## 5 Discussion

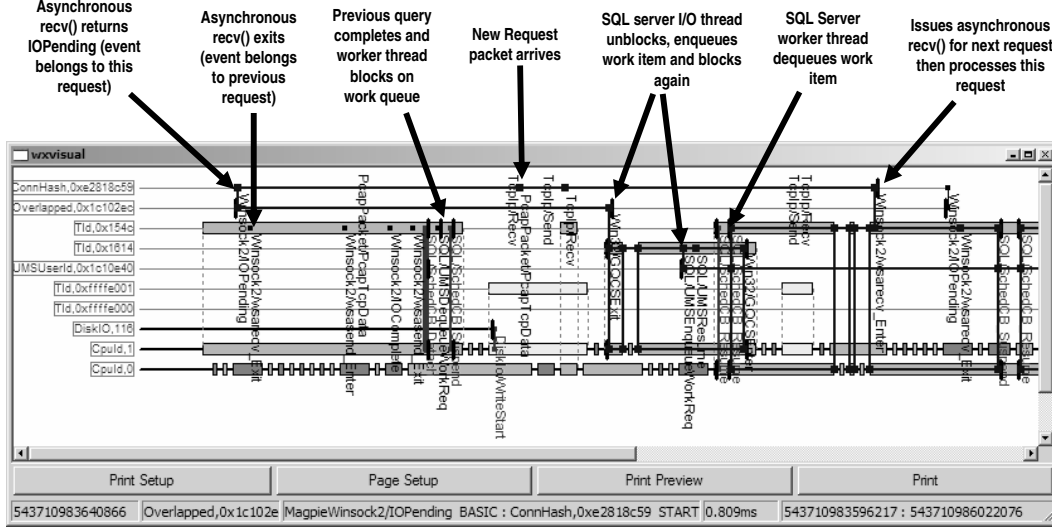
Although conceptually simple, we have found in practice that there are a number of subtleties in the defining of the event schema. In particular, asynchronous I/O can be quite challenging to get right<sup>2</sup>.

Figure 4 contains an annotated screenshot from an automatic visualization tool for our request parser, showing a typical excerpt from an SQL Server request. The challenge is that a thread (*Tid*=0x154c in the diagram) enqueues an asynchronous receive for the next request before it has finished processing the current request. The key to expressing this behavior in the schema is the Winsock2/IOPending event which we post when the asynchronous invocation returns (but before the I/O completes). The schema associates the *Overlapped* and *ConnHash* attributes starting with the Winsock2/IOPending event and finishing with the Win32/GQCSExit event. This ensures that any received packets are collected correctly. The *ConnHash* attribute is invalidated for the current request at the subsequent Winsock2/wsarecv\_Enter event.

In spite of the challenges of defining an event schema

---

<sup>2</sup>On Windows, asynchronous or overlapped I/O can be performed on any file handle. Most API calls accept an optional argument of type OVERLAPPED which is used as a cookie to associate requests with completions. If the I/O request is not satisfied immediately then STATUS\_IO\_PENDING is returned. Completion of the request may be signalled via a number of mechanisms including NT Events and Completion Ports.



**Figure 4.** Screenshot from request parser visualization tool showing detail of asynchronous socket receive in an SQL Server request.

in some cases, particularly when the application aggressively pipelines I/O, we have successfully parsed not only HTTP requests but also SQL Server requests under a TPC-C workload. As well as the asynchronous I/O pattern described above, SQL Server’s own scheduler multiplexes its concept of “users” onto operating system threads, minimising interactions with the OS scheduler by providing a set of user-level synchronization primitives. We therefore regard SQL Server as a particularly demanding proof-of-concept application which demonstrates the feasibility of our temporal join and event schema techniques.

## 6 Related work

Magpie has used two very different ways of tracking requests through a system. Originally, a globally unique identifier was assigned when a request arrived and propagated from one component to another. Events (annotated with this identifier) were then logged by each component. Alternatively the request path can be inferred by postprocessing the event stream and correlating related events. This second approach has many advantages in terms of scalability, flexibility, backwards compatibility and support for heterogeneous components. For these reasons, Magpie now uses the event correlation technique presented here.

Pinpoint [4] tracks requests using global identifiers propagated using modified middleware components. The focus is on fault detection and localization, and so

Pinpoint does not currently monitor resource consumption by requests. It takes advantage of a homogenous middleware layer throughout the system which simplifies the identifier propagation mechanisms.

Aguilera *et al.* [1] correlate message trace events to infer causal paths and to locate the nodes which cause performance bottlenecks in a distributed system. They present two algorithms both of which use statistical methods to identify communication patterns and hence sources of latency. In contrast, both Magpie and Pinpoint follow each and every request through multiple software components within machines, as well as across network connections.

We borrow the term “temporal join” from the temporal database field [5], and indeed, if we stored our events in such a database, then running temporal outer join queries for each request time interval should in theory achieve the same results. The use of START/STOP binding barriers on event attribute values is the same as defining a valid-time interval on a particular tuple. To allow the expression of more complicated transitive associations between bindings we specify how they are modified as part of the join operation, whereas in a database this could be achieved with triggers. It would be interesting future research to try and express our schema semantics in a temporal query language, although it is not likely that this would clarify the exposition.

The problem of generic request extraction has some similarities to the problems addressed by *continuous query* databases such as Tapestry [9] and Telegraph [3]. These are databases where users express queries whose result

sets appear as if the query were executed continuously over time. The online variant of our Magpie request parser faces similar issues.

## 7 Conclusion

A great advantage to using an event schema is the ability to reuse the same instrumentation and parser to extract many different types of request. In contrast, propagation of a unique request identifier can result in complications when the request definition changes. In the two-tier web site a single HTTP request can generate many nested database RPCs, while in a different application such as TPC-C the database transaction is itself a complete request.

It is not feasible when instrumenting a component to have to consider in advance all the ways in which that component will be deployed in the context of a “request”. Nor is it desirable to have multiple versions of instrumentation to support multiple applications. Furthermore, the event schema allows us to develop a single parameterized tool for request parsing, rather than having to build bespoke request extraction tools for each different application.

Support for generic request extraction from live event streams is part of a bigger picture for system self-configuration and maintenance. We encourage operating system and middleware developers to provide such instrumentation as a matter of course, allowing applications to develop their own schemas which reflect how they use system resources. There should be no need for the complexity and overheads of propagating globally unique request identifiers, no need to write application-specific request parsers, and instrumentation re-use should be straightforward. The vision of self-managing systems will only be realised if we give serious consideration to developing generic, flexible mechanisms for end-to-end request tracking.

## References

- [1] 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)*, pages 74–89, Oct. 2003.
- [2] P. Barham, R. Isaacs, R. Mortier, and D. Narayanan. Magpie: online modelling and performance-aware systems. In *9th Workshop on Hot Topics in Operating Systems (HotOS IX)*, pages 85–90, May 2003.
- [3] S. Chandrasekaran, O. Cooper, A. Deshpande, M. Franklin, J. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Proceedings of Conference on Innovative Data Systems Research (CIDR)*, Jan. 2003.
- [4] M. Chen, E. Kiciman, E. Fratkin, E. Brewer, and A. Fox. Pinpoint: Problem determination in large, dynamic, Internet services. In *Proceedings of the International Conference on Dependable Systems and Networks (IPDS Track)*, pages 595–604, June 2002.
- [5] D. Gao, C. S. Jensen, R. T. Snodgrass, and M. D. Soo. Join operations in temporal databases. Technical Report TR-71, TIME-CENTER, Oct. 2002.
- [6] J.O.Kephart and D.M.Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, Jan. 2003.
- [7] I. Park and M. K. Raghuraman. Server diagnosis using request tracking. In *First workshop on the Design of Self-Managing Systems, held in conjunction with DSN 2003*, June 2003.
- [8] M. Stonebraker, L. A. Rowe, and M. Hirohama. The implementation of postgres. *IEEE Trans. Knowledge and Data Engineering*, 2(1):125–142, 1990.
- [9] D. Terry, D. Goldberg, D. Nichols, and B. Oki. Continuous queries over append-only databases. In *Proceedings of the 1992 ACM SIGMOD international conference on Management of data*, pages 321–330. ACM Press, 1992.