Singularity Design Note 14

Tracing

Logging API and instrumentation guidelines

Tracing by means of logging events is a powerful tool for monitoring and diagnostics. Flexibile, fine grained and low cost tracing is useful for a large range of applications, ranging from debugging to performance analysis and anomaly detection. This design note proposes that Singularity should provide an efficient and dynamic logging infrastructure, and that instrumentation should be ubiquitous across operating system components and applications. It defines a logging API and suggests instrumentation guidelines for developers.

1. Introduction

Comprehensive system instrumentation enables the following two key capabilities:

1. the attribution of specific instances of resource consumption to a particular sequence of actions belonging to a program; and
2. the observation of causal relationships between and within programs.

This information can be applied in many ways, including performance analysis and debugging, functional debugging, self-tuning, workload modelling and anomaly detection. It also supports Singularity's dependability goal by providing for inspection of the state and progress of the running program in a manner which can not only reveal what is actually happening, but may also explain why.

1.1. Terminology

As defined in *SDN10: Resource Management and Scheduling*, a **task** is "an individual application step or sequence of actions that the program desires to be completed by a specified deadline". In this document we use the notion of tasks without deadlines to define the unit of interest being traced, referred to as an **activity** (the same terminology is used by the Windows End2End Tracing Taskforce). An activity can be any sequence of related actions, and may incorporate several tasks. In the literature, an activity is sometimes referred to as a **request**.

1.2. Motivation

Some uses and benefits of activity tracing in Singularity include the following.

- In order to support soft real-time guarantees, it is planned that Singularity will perform multi-resource scheduling based on deadlines and resource reservations. Effective coordination of the various schedulers is dependent on knowing the causal relationships between the different resource demands within a task. Although tracing will only reveal this information historically, it might be used to debug the scheduler(s), to automatically infer appropriate resource reservations for a given workload based on past behaviour, to dynamically break a task into sub-tasks to optimise scheduling and to perform admission control.

- The perception of dependability in an operating system by the user is often based on the predictability of response times. The audit trail created by activity tracing is extremely useful for performance debugging as it can reveal unexpected interactions or problems which occur across independent components. For example, it might be used to determine why an action such as opening a folder in an explorer window can sometimes be very fast and other times hang for no apparent reason.
- Activity tracing is also useful for exposing the effects of algorithms used to control access to a shared resource. For example, instrumentation of buffer cache hits, misses and evictions can allow a systematic tuning for performance according to the characteristics of the actual workload experienced on a system.
- By monitoring the behaviour and resource consumption of individual activities, accurate models of the live system workload can be constructed and used for performance analysis tasks such as prediction and capacity planning.

Note that the information exposed by tracing application activities across multiple system components is complementary to that revealed by per-component monitoring. For example, a sudden increase in queue length might be caused by a local fault, and this could be identified by looking at performance of just that component over time. Alternatively, the queue length fluctuation could be a result of a change in the characteristics of the system workload – a different mix of activity types may exercise other parts of the system which are slow or faulty and cause a bottleneck. If only a subset of the workload is responsible for triggering the fault, while the entire workload suffers the consequences, then it can be very difficult to isolate the cause of the problem without knowing the dependencies within activities.

2. Event logging infrastructure

The event logging infrastructure provided by the operating system will have a conventional, decoupled producer/consumer structure, with a controller to manage the delivery of events from producers to consumers. This has several advantages for flexible and targetted tracing:

- Fine-grained and dynamic control over the set of events which are being generated at any time.
- The mapping of producers to consumers can be arbitrarily many-to-many.
- The facility to filter events close to the source. At a minimum, the logging infrastructure will support different logging levels, but could conceivably deploy more sophisticated event filters by inserting proxy event consumers.

This design note aims to establish guidelines such that event consumers can easily trace activities end-to-end, according to a self-determined definition of activity as appropriate. The way in which event consumers handle and interpret events is beyond the scope of this document, however with the right supporting infrastructure and sufficient instrumentation, correlation of events in order to extract activities will be straightforward.

2.1. Requirements

This section discusses ideal and possible requirements for the logging infrastructure.

2.1.1. High performance

The infrastructure must support the logging of every single operating system action with minimal performance penalty. Compact event formats, circular in-memory buffers and delayed flush to disk are probably essential.

2.1.2. Low overhead

It should be possible to have always-on logging, and online, real-time event consumption. As a baseline figure, event generation on Windows Server 2003 is claimed to introduce a 5% CPU overhead for a sustained rate of 20,000 events/sec, and experience with logging every context switch on a loaded system bears this out as accurate. Although high performance is not a priority for Singularity itself, it would be sensible to quantify the disk and network (if any) overheads of the logging infrastructure also.

2.1.3. Dynamic

Individual event producers and consumers can be enabled and disabled dynamically. A disabled event producer, or a producer with no consumers, should have negligible performance impact. Events can be written to disk and consumed offline, or consumed online in real-time.

2.1.4. Ubiquitous

Tracing an activity across domain boundaries should be seamless. There should be no arbitrary exclusion of any operating system component or driver.

2.1.5. Supporting sampled tracing

In some situations the cost of tracing every activity might be prohibitive, or it may simply not be necessary to capture every single one. Support for sampled tracing can be provided by exploiting the mechanism by which activities span multiple processes and providing an in-band control mechanism in addition to the standard out-of-band control.

2.1.6. Secure

Securing access to trace data should be addressed when the security model is finalised. As is the case for debugging, by exposing the internal workings of the system, tracing inevitably compromises its security. Issues will include:

- The revealing of execution information which applications may wish to conceal.
- Securing access to trace logs.
- The fact that not all event generating components are equally trusted, and consequent vulnerabilities to denial of service attacks etc.

2.1.7. Reliable

There are obvious performance trade-offs in ensuring reliable delivery of events. It may be appropriate to distinguish events which must never be dropped, and those for which best-effort is good enough.

2.1.8. Verifiable

A static tool to determine logging coverage would be useful for setting the expectations of users of tracing data, by revealing which parts of the system are "blind spots". Such a tool could also be used as a mechanism for ensuring that Singularity and its applications are comprehensively instrumented, as it is not always easy to manually check this kind of source-level annotation.

Clearly a tool which could statically verify logging coverage could also insert the instrumentation itself, and this may indeed be the best way to instrument operating system synchronization and resource usage. Developers would still need to log higher-level causal relationships and insert other informational events which are not amenable to static checking.

Verification of logging coverage, and automatic instrumentation is a desirable but not essential requirement. In the discussion following, especially the usage guidelines, we assume this kind of tool will not exist.

2.2. Implementation issues

2.2.1. Timestamps

In order to observe causal relationships at the granularity of synchronization objects, there must be a strict ordering on events. Event timestamps should be the highest precision possible (eg cycle counter).

2.2.2. Event correlation

For correlating events across domains, every object which is dispatched onto a queue must have an identity. It must be possible to match the sending of a message with its subsequent receipt, or to associate a buffer cache request with a specific hit or miss.

2.2.3. In-band trace control

To support in-band trace control, every message has a trace bit which can be set on or off. Every component which receives or transmits messages must include logic to propagate the trace bit.

2.2.4. Shared resources

It is often not known a priori how to account resource that is multiplexed among several entities. However if the instrumentation is sufficiently detailed, this is a late binding decision which can be made according to the specific goals of the event consumer.

2.2.5. Event storage

An important implementation choice is where to buffer events and who owns the storage. Having a buffer per consumer allows early event filtering and manipulation (eg combining multiple events into one), but makes it harder to ensure reliable event delivery. If there is a buffer per producer, then multicast of events to several consumers is more efficient, but there is less flexibility.

2.2.6. Metadata and naming

Event formats should be defined once in a single place. The details of naming and versions of events, producers and consumers, are currently to be determined.

2.2.7. Propagation of activity tokens

It may be possible to propagate tokens in-band which uniquely identify activities (or sub-parts of activities), and to annotate every event with the appropriate token. This has advantages for robustness in the face of dropped or missing events and for simplicity of post-processing the event logs. One implementation alternative which could be considered is to propagate tokens via the type system.

The drawbacks of this approach are the difficulty of determining upfront the unit of granularity over which an activity is uniquely identified. An activity for one application may comprise a single database transaction, while for another it consists of several transactions invoked by a single stored procedure. To be absolutely general, an activity token should be propated over the smallest possible number of events, just sufficient to ensure that related events can be correlated. However undoubtably atomic regions of execution which will always be treated as belonging to a single activity can be identified, and propagation of an activity token is a sensible optimisation at these points.

3. Proposed API

3.1. Logging control (out-of-band)

```
EnableLogging(ProducerId, LogLevel)

DisableLogging(ProducerId, LogLevel)
```

3.2. Logging control (in-band)

Additional methods to be defined on the message object to get, set and clear trace bit.

3.3. Registration

```
RegisterProducer(ProducerId)

RegisterConsumer(ConsumerId, list_of_ProducerIds, Callback)
```

3.4. Event generation

```
PostEvent(ProducerId, LogLevel, EventId, EventArgs)
```


4. Usage guidelines

Certain events are critical for activity tracing to be possible, while others are informational. Critical events are essential for event correlation and allow resource usage to be attributed to activities. These events are posted at the following places:

1. Synchronization. An event should be posted on every channel send and receive, logging the identity of the message. If there are locks in Singularity, these should also be instrumented (ie requests for the lock, acquisitions and releases). This allows the causality within an activity to be observed.
2. Transfer of control. Whenever a different thread or process takes over this should be logged, with some correlation id. This allows the progress of the activity to be traced.
3. Physical resource consumption. Every access of physical resource should be logged with sufficient information to attribute the resource consumption to an individual activity. Physical resources include disk, network, CPU, power, memory. This indicates the service process received by the activity, as is required for diagnostic tracing.
4. Virtual resource consumption. Physical resource contention is often arbitrated through some virtual resource such as cache, network connection, file system or scheduler. Every access to the virtual resource should be logged. Some access requests will be explicit, eg a cache lookup, others will be implicit, eg thread becomes runnable. All this information is useful for inferring the service process of the activity, as required for workload modelling.

4.1. Operating system developers

As much as possible, critical events should be generated by operating system components. In particular, synchronization and control transfer primitives should all incorporate instrumentation, as well as whenever any shared resource is multiplexed. So for example, every channel send and receive operation should be instrumented.

Ideally the operating system will also provide support for propagating the in-band trace bit.

4.2. Application developers

As well as adhering to the usage guidelines above, application developers would help ensure comprehensive instrumentation by favouring operating system synchronization primitives which are already instrumented. For example, by using the provided thread pool constructs rather than writing new ones.

4.3. Logging levels

The API provides for selective enabling or disabling of specific logging levels, and every event is posted with an identifying level or levels. In order to ensure that logging levels are consistent system-wide, the following flags can be combined arbitrarily both to describe which levels an event belongs to, and which levels an event consumer would like to receive..

```
TRACE_NONE              // nothing logged
TRACE_CONTINUOUS        // log start/end activity, suitable for continuous background monitoring
TRACE_SYNCHRONIZATION   // log synchronization
TRACE_CONTROLTRANSFER   // log transfer of control
TRACE_RESOURCEDEMAND    // log physical resource consumption
TRACE_VIRTUALDEMAND     // log virtual resource demands and service
TRACE_INFORMATIONAL     // log other application-specific information
```

For example, an event recording the start of an activity will be flagged as both TRACE_CONTINUOUS and TRACE_INFORMATIONAL. An event consumer monitoring caching effects on activities will enable logging at the TRACE_RESOURCEDEMAND and TRACE_VIRTUALDEMAND levels.

