# Causal Tracing for Go

Joshua Liebow-Feeser

## 1. Introduction

Brown University Computer Science's Tracing Framework is a collection of applications aimed at various tasks in creating, analyzing, and debugging distributed systems. The core of the framework is the Tracing Plane, which contains components common to all of the applications in the framework. One of these components is *Baggage*, a simple metadata format that, when propagated along the path of a request, allows for powerful analysis. This is part of a broader pattern among recent distributed systems work and research in using metadata propagation to gain insight into the runtime execution of a distributed system.

In order to fully leverage the benefits of metadata propagation, three facilities are required:

1. The ability to send a set of metadata along with a cross-machine request such as an RPC or an API call
2. The ability to carry this metadata along with the execution of the request handling so that it is available to be further propagated if the request handling code initiates further RPC or API calls
3. The ability to have all code called by the request-handling code access this metadata

The Brown Tracing Framework is implemented in Java, and takes advantage of certain features and properties of Java to provide these three facilities. For example, Java's support for thread-local storage allows for metadata for a request to be stored local to the thread that handles the request. That way, it can be retrieved and attached to any further RPC or API calls that are generated during request handling. Further, Java's runtime code modification support allows the Baggage to be extracted from and inserted into RPC and API requests that are implemented without knowledge of Baggage.

In this project, I extend these facilities to Go. Go differs significantly from Java in a few key ways that are relevant to these facilities:

- Go does not provide thread-local (or, in Go parlance, goroutine-local) storage
- The Go community discourages the use of implementations whose behavior is not obvious from source code. This increases readability and reduces the amount of time it takes to become familiar with a new codebase, but it also means that automatically modifying RPC or API calls would be considered bad style.
- Go does not support runtime code modification

While it may seem that these differences would make it more difficult to implement the metadata facilities in Go, in some ways, it actually makes it easier. Because of Go's emphasis on explicit behavior, the tracking of a request's state is standardized in the Go standard library's context package as of Go 1.7. According to the documentation, this package "defines the Context type, which carries deadlines, cancelation signals, and other request-scoped values across API boundaries and between processes." The documentation lays out explicit rules for using the context.Context type - it must be explicitly passed as the first parameter to every function invoked in the handling of a request. The use of this context.Context object, now standardized across the standard library, and becoming more standard in other major Go projects, solves facility (2) described above.

As for facility (1), the recently-released gRPC RPC framework from Google handles this natively in Go with their metadata package, providing the ability to send metadata in an RPC request and extract metadata in an RPC handler.

This leaves facility (3), which is what this project implements. The difficulty of this problem is that not all function calls in all packages accept a context.Context argument. For some, this is a matter of backwards-compatibility. For others, it's a matter of clutter - it's best to avoid an extra parameter that will almost never be used. However, some of these functions may need to inspect or modify the metadata pertaining to the request being served by their parent functions. For example, the Tracing Framework's X-Trace system works by having each logging call access request-local metadata to determine where it fits in a global execution graph that spans multiple processes and even multiple machines. In order to add this functionality to an existing system that uses normal, context-unaware logging calls, facility (3) is required.

Briefly, this is achieved by modifying the Go runtime to support goroutine-local storage, and providing a tool which rewrites Go source code to propagate this goroutine-local storage when a new goroutine is created.

The rest of this report is laid out as follows. In section 2, I describe the modification of the runtime to support goroutine-local storage. In section 3, I describe the tool that I created to rewrite Go source code to propagate storage between goroutines. In section 4, I describe the local package that I wrote to provide a high-level interface to this storage. In section 5, I describe the adaptation of a logging system to use these facilities to implement X-Trace.

## 2. Goroutine-Local Storage

*Source: current version; version as of this paper's writing; see* `src/runtime` *under the directory for your architecture.*

Go's design philosophy of simplicity lead the designers to discourage the use of code which relies on the identity of a goroutine. Not only is goroutine-local storage not supported, but goroutine IDs are not even made available to the programmer (although clunky, expensive workarounds do exist). In order to support goroutine-local storage, I had to modify the runtime to support it.

In the runtime, every goroutine has associated with it a structure called a “g” (defined in runtime2.go) which is a collection of goroutine-specific fields such as pointers to the goroutine’s stack beginning and end, the goroutine’s ID, and a description of which OS thread the goroutine is scheduled in. Since Go is self-hosting, this “g” type is a native Go struct type.

Adding goroutine-local storage involved first adding a new field - “local” of type interface{} - to the g type, and second creating getter and setter functions - GetLocal() interface{} and SetLocal(local interface{}). When a new goroutine is spawned, its newly-allocated g has its local field automatically set to nil, ready to use.

## 3. Rewrite Tool

*Source: current version; version as of this paper’s writing*

In addition to goroutine-local storage, a necessary component is the propagation of metadata from one goroutine to another when a goroutine is first created. In order to facilitate this propagation, I created a tool which rewrites Go source code so that all new goroutines automatically inherit a copy of the creating goroutine’s goroutine-local storage.

For example, imagine that a program’s source code contains the following creation of a new goroutine:

```
1: go doSomething(arg1)
```

The source code rewriting tool would rewrite that into:

```
1: go func(__f1 func(), __f2(arg1 int), arg1 int) {
2:    __f1()
3:    __f2(arg1)
4: }(local.GetSpawnCallback(), doSomething, arg1)
```

The key difference between this generated code and the original is the use of `local.GetSpawnCallback`, which creates a closure that, when invoked on line 2, initializes the new goroutine’s local storage based on the state of the parent goroutine’s local storage at the time of creation. This is explained in more detail in section 4.

## 4. `local` package

*Source: current version; version as of this paper’s writing; documentation*

On its own, the goroutine-local storage that I implemented in the runtime is a very low-level feature - it only supports storing a single interface value. Package `local` provides a higher-level interface to goroutine-local storage. In particular, it allows for multiple different pieces of code to simultaneously use goroutine-local storage without conflicting with each other. In addition, it allows for the initial local storage of a given goroutine to be initialized using arbitrarily complex logic rather than simply a copy of the parent goroutine's local storage.

The separation of different distinct variables stored inside a single goroutine-local storage instance is achieved through the use of opaque "tokens" (which, at the time of writing, are simply integer indexes into a slice of values). A consumer of the local package requests its own token by calling Register, setting both the default initial value for its local variable, and a set of callbacks that are used to control certain behavior such as how a new goroutine's local storage is initialized based on its parent's local storage.

```
// Register registers a new local variable whose
// initial value and callbacks are set from the
// arguments. The returned Token can be used to
// identify the variable in future calls.
//
// Register should ONLY be called during initialization,
// and in the main goroutine
func Register(initial interface{}, c Callbacks) Token
```

## 5. X-Trace Implementation

*Source: current version; version as of this paper's writing; documentation*

X-Trace is a "logging framework for distributed systems." Execution and causality are tracked using unique identifiers and causal relationships between those identifiers. Each time a log line is emitted, it is tagged with the thread's current identifier, a new identifier is generated, and the relationship between the identifiers - that the former causally preceded the latter - is emitted along with the log line itself. When taken together, these pieces of data are sufficient to reconstruct a full execution graph, adding meaningful structure to an otherwise relatively unstructured collection of logs.

In order to test my implementation of goroutine-local storage and propagation, I used it to implement a client for X-Trace logging in Go. The client is able to assign unique identifiers and causal relationships between identifiers for every log line in a program, and propagate these identifiers across RPC boundaries using a wrapper around the Go `grpc` package.

## Local Propagation

The propagation of identifiers in X-Trace maps very cleanly onto the propagation of goroutine-local storage that I implemented. With a given goroutine, a unique “Event ID” (in X-Trace parlance) is kept in local storage, along with a “Task ID” that identifies the overall computation that the given event is a part of. When a log line is emitted, the current Event ID is considered to be its parent, a new Event ID is generated for the log line itself, and the causal relationship between the parent and child Event IDs is recorded. Finally, the child Event ID becomes the new current Event ID, overwriting the parent in local storage.

In addition to tracking causality in a single thread of execution, X-Trace also requires tracking causality when new threads are created. Thus, the Go X-Trace implementation must support tracking when new goroutines are created. This is as straightforward as one might imagine - simply use the parent goroutine’s Task and Event IDs to initialize the child goroutine’s IDs.

In the entire client implementation, the only functions that require knowledge of the existence of goroutine-local storage are the getters and setters for Task and Event IDs:

```
func GetTaskID() (taskID int64) {
  return getLocal().taskID
}
```

```
func SetTaskID(taskID int64) {
  getLocal().taskID = taskID
}
```

```
func GetEventID(eventID int64) {
  return getLocal().eventID
}
```

```
func SetEventID(eventID int64) {
  getLocal().eventID = eventID
}
```

## Remote Propagation

*Source: current version; version as of this paper’s writing; documentation*

In order to propagate X-Trace Task and Event IDs along the path of RPC requests, I wrote a package which wraps certain functions of the standard `grpc` package. These wrappers automatically extract and propagate Task and Event IDs using the getters and setters exposed by my X-Trace `client` package.

This package provides two functions: `Invoke` wraps the standard `grpc` package's `Invoke` function, but extracts and propagates Task and Event IDs. Existing source code can simply call this instead of the standard `Invoke`, and remain otherwise unmodified. `ExtractIDs` extracts Task and Event IDs from the current `context.Context`'s gRPC metadata, and inserts them as the current goroutine's Task and Event IDs. Existing code needs to be modified to call `ExtractIDs` at the beginning of each gRPC handler.

In order to test remote propagation, I wrote an example service (current version; version as of this paper's writing). It performs simple mathematical functions by chaining together RPC calls. Each instance of the service is responsible for a single unary operation (for example, "add 5 to the argument"), and for passing the result of its operation to the next service for further processing. Here's a sample invocation:

```
$ # When RPC calls are received on :1234, add 7 to the argument,
$ # and pass the result to the RPC handler at :5678
$ ./example --xtrace-server :5563 --add 7 --listen-addr :1234
--next-server :5678

$ # In a separate window, run this: when RPC calls are received
$ # on :5678, multiply the argument by 3, and return the result
$ ./example --xtrace-server :5563 --mul 3 --listen-addr :5678

$ # Compute (3 + 7) * 3 by passing the argument 3 to the RPC handler
$ # at :1234
$ ./example --xtrace-server :5563 --next-server :1234 --in 3
Got value 30
```

As a result of this sequence of invocations, the following events are recorded by the X-Trace server:

```
task_id: 5655435119442465856
event_id: 6289766621339003385
parent_event_id: 5470210360033630164
timestamp: 1474851945531432
process_id: 40208
process_name: "./example --xtrace-server :5563 --next-server :1234
--in 3"
label: "passing --in (3) to :1234"
```

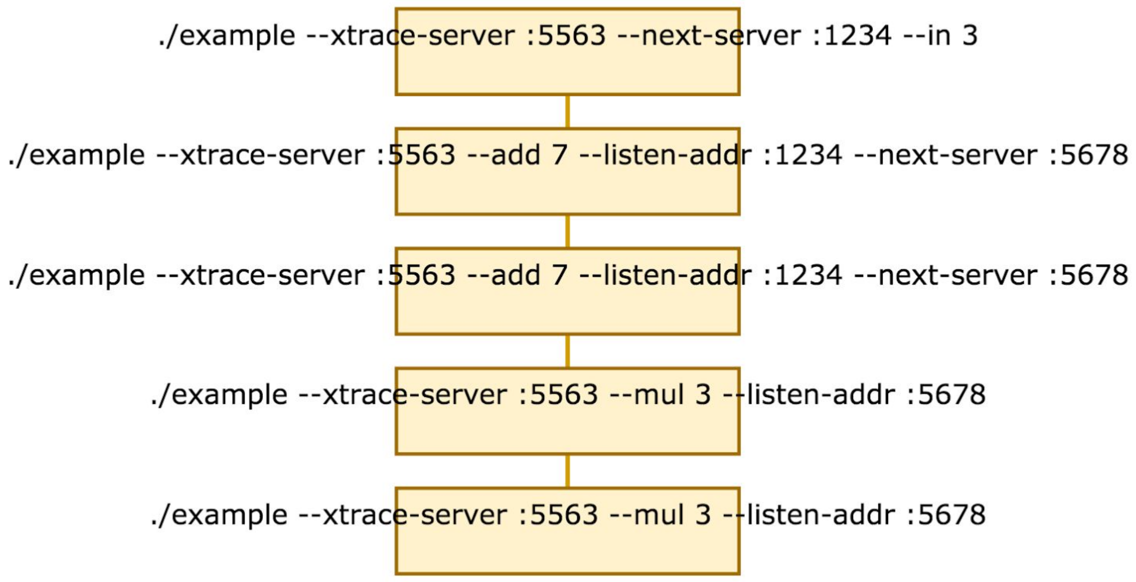
```
task_id: 5655435119442465856
event_id: 8768524173443383991
parent_event_id: 6289766621339003385
timestamp: 1474851945538647
process_id: 40207
process_name: "./example --xtrace-server :5563 --add 7 --listen-addr
:1234 --next-server :5678"
label: "applying operation +7 to argument 3: 10"


task_id: 5655435119442465856
event_id: 3621238959106384114
parent_event_id: 8768524173443383991
timestamp: 1474851945538780
process_id: 40207
process_name: "./example --xtrace-server :5563 --add 7 --listen-addr
:1234 --next-server :5678"
label: "passing to :5678"


task_id: 5655435119442465856
event_id: 2500724463162741905
parent_event_id: 3621238959106384114
timestamp: 1474851945540604
process_id: 40204
process_name: "./example --xtrace-server :5563 --mul 3 --listen-addr
:5678"
label: "applying operation *3 to argument 10: 30"


task_id: 5655435119442465856
event_id: 2128074912784557060
parent_event_id: 2500724463162741905
timestamp: 1474851945540877
process_id: 40204
process_name: "./example --xtrace-server :5563 --mul 3 --listen-addr
:5678"
label: "returning value 30"
```

Visualizing this causal graph, we get the following. Each event is annotated with the process name that emitted the event (and, in the web UI that produces this graph, hovering over a given event shows all of the details given in the text log above).


./example --xtrace-server :5563 --next-server :1234 --in 3
./example --xtrace-server :5563 --add 7 --listen-addr :1234 --next-server :5678
./example --xtrace-server :5563 --add 7 --listen-addr :1234 --next-server :5678
./example --xtrace-server :5563 --mul 3 --listen-addr :5678
./example --xtrace-server :5563 --mul 3 --listen-addr :5678

Note that because no goroutines are spawned during the handling of an RPC call in this example service, the rewrite tool described above was not used.

## Integrating Local Propagation with Existing Code

Note that the use of getters and setters for Task and Event IDs is not limited to RPC calls. If, for whatever reason, some subsystem were to be invoked with a `context.Context` argument that carried with it Task and Event IDs, setters could be used to propagate the IDs as though they were received from an RPC call. Similarly, if a subcomponent expected IDs stored in a `context.Context` argument, getters could be used to extract the current IDs for propagation through the `context.Context`.