The Design of Transparent Telemetry

Transparent Telemetry, Part 2

Russ Cox

February 8, 2023

research.swtch.com/telemetry-design

I believe open-source software projects need to find an open-source-friendly way to do telemetry. This post is part of a short series of posts describing *transparent telemetry*, one possible answer to that challenge. For more about the rationale and background, see the previous post*. For additional use cases, see the next post*.

Transparent telemetry is made up of five parts:

- Counting: Go toolchain programs store counter values in per-week files maintained locally.
- Configuration: There is a reviewed public process for defining a new graph or metric to track and publish on the Go web site. The exact counters that need to be collected, along with the sampling rate needed for high accuracy results, are derived from this configuration.
- Reporting: Once a week, an automated reporting program randomly decides whether to fetch the current configuration and then whether to be one of the sampled systems that week. If so, it reports the counters listed in the configuration to a server run by the Go team at Google. In typical usage, we expect a particular Go installation to report each week with under 2% probability, meaning less than once per year on average. [*Update*, 2023-02-24: The design has been changed to be opt-in*, which raises the expected reporting probabilities.]
- Publishing: The server publishes each day's reports in full (in a compressed form) as well as publishing the tabular and graphical summaries defined in the configuration.
- Opt-out: The system is enabled by default, but opting out is as straightforward, simple, and complete as possible. [*Update*, 2023-02-24: The design has been changed to be opt-in*.]

This post details each of these parts in turn. I wrote an implementation of local counter collection to convince myself it could be made cheap enough, but as of the publication of this post, no other part of the system exists today in any form. I hope that the system can be built for Go over the course of 2023, and I hope that other open-source projects will be interested to adopt this approach or inspired to explore others.

Counting

Go toolchain programs (`go` and the other programs that ship in the Go distribution, like `go tool compile` and `go tool vet`, along with other Go team-maintained programs like `gopls` and `govulncheck`) collect counter values in local files using a simple API:

```
package counter

func New(name string) *Counter
func (c *Counter) Inc()

func NewStack(name string, frames int) *Stack
```

```
func (s *Stack) Inc()
```

Basic named counters are created with `counter.New`, typically assigned to a global variable (this has no init-time overhead), and then incremented as the program runs by using the `Inc` method.

For example, suppose we want to monitor the typical build cache miss rate for a `go build` command. Each `go` command invocation can track the miss rate during its run and then increment one bucket of a histogram with exponentially-spaced buckets: 0%, <0.1%, <0.2%, <0.5%, <1%, <2%, <5%, <10%, <20%, <50%, and <100%. After a week, those 11 counters record the distribution of build cache miss rate experienced on that system.

Stack counters are similar, but the constructor also takes the maximum number of frames to record. Each frame is represented by an import path, function name, and line number relative to the start of the function, such as `cmd/compile/internal/base.Errorf+10`. The counter name is the concatenation of the the name passed to the constructor and the given number of frames. For example, the counter name for one increment of the result of `NewStack("missing-std-import", 5)` might be

```
missing-std-import
cmd/compile/internal/types2.(*Checker).importPackage+39
cmd/compile/internal/types2.(*Checker).collectObjects+54
cmd/compile/internal/types2.(*Checker).checkFiles+18
cmd/compile/internal/types2.(*Checker).Files+0
cmd/compile/internal/types2.(*Config).Check+2
```

A line number relative to the start of the function is fairly stable across unrelated edits in the source code, making it possible to identify the same stack trace even across different versions of a program.

One of the key properties of transparent telemetry is that uploaded reports only contain strings that are already known to the collection server. Using an import path instead of the full file path allows aggregation across different systems and more importantly avoids exposing details like the full path to a directory where the Go compiler source code was stored when it was built. Another important consideration is that function names in modified copies of Go tools might contain unexpected strings: we don't want to know about a modified copy that adds `types2.checkWithChatGPT` to the call stack. This problem is handled by only saving stack traces from specific unmodified, released versions of tools. The version information and the presence of any modifications can be identified using the build information embedded in the binary*. (In contrast, .NET reports full file system paths and then documents that it is the developer's responsibility to "avoid inadvertent disclosure of information*" by not building the software in "directories whose path names expose personal or sensitive information". The burden of not exposing private data in a telemetry system should never be placed on users.)

The counter files are stored in the directory `<user>/go/telemetry/local/`, where `<user>` is the user configuration directory, as reported by `os.UserConfigDir*`. Each file is named by the program's name, version, build toolchain version, GOOS, and GOARCH, along with the date of the start of the week. For example:

```
/Users/rsc/Library/Application Support/go/telemetry/local/
    compile-go1.21.1-darwin-arm64-2023-01-04.v1.count
    gopls@v0.11.0-go1.21.1-linux-386-2023-01-04.v1.count
    ...
```

The version and build toolchain version are recorded only as `devel` for unver-

sioned tools, such as when developing Go itself.

Aggregating counters by week has two important purposes. First, it should help reduce privacy concerns by making clear that there is no way to reconstruct any kind of fine-grained trace of user behavior. Second, reporting counters by week reduces statistical noise caused by persistent usage variations such as weekends. Every long-term monitoring dashboard I have ever seen begins by computing 7- or 28-day averages of the data to remove this high-frequency noise. Removing it client-side gives both more privacy and cleaner reports.

When Go telemetry first creates the `local` directory, it randomly selects the start of that system's week. The system in our example has chosen weeks beginning on Wednesday. The random choice of week start spreads the server load over the week and also provides prompt reporting of new problems: if a new Go distribution is published on Tuesday, one seventh of the systems that install it immediately will include Tuesday's operation in the Wednesday uploads.

These files use a custom binary format that starts with a simple key-value header repeating the information that went into the file name:

```
Week: 2023-01-04
Program: compile
GoVersion: go1.21.1
GOOS: darwin
GOARCH: arm64
```

After the header come the counters, in an on-disk hash table suitable for memory mapping into each running instance of the program. Those instances use lock-free atomic operations to increment counters and maintain the file, keeping the overhead associated with telemetry very low. The design also avoids any possibility of deadlock or unbounded latency when a counter is incremented, even if one instance of the program is hung or otherwise misbehaving. A tool, perhaps called `go tool telemetry`, will convert one of these binary files to JSON for processing by other interested programs.

Note that the raw data stored on disk is only names of counters and their associated 64-bit totals. There is no event log or any more detailed kind of trace. The decision to maintain the counters directly, instead of deriving them from a more detailed trace, is motivated mainly by concerns about disk space and update latency. However, never having any kind of event log or trace also reduces the privacy impact of the local collection.

A local web server (perhaps `go tool telemetry -http`) will display the local counters and be able to graph counter data over time for user inspection (at only 1-week granularity, of course).

Configuration

Data collection in transparent telemetry starts with the reason the data is being collected: a specific graph that is going to be computed, along with the specific margin of error desired for that graph. From that graphing configuration, the transparent telemetry server can compute the reporting configuration, declaring which counters to report at what sampling rate in order to produce that graph.

For example, a graphing configuration for the Go build cache miss rate graph we considered in the previous section might look like:

```
title: Go build cache miss rate
type: histogram
error: 1%
counter: go/buildcache/miss:{0,0.1,0.2,0.5,1,2,5,10,20,50,100}
```

For a margin of error of 1% at a 99% confidence level, we need about 16,000

samples*. The server would keep track of an estimate of the number of reporting systems and adjust the sampling rate each week to produce the right number of samples. If there are one million reporting systems, then the sampling rate to get 16,000 samples is 1.6%, so the corresponding reporting configuration would sample each counter with that probability.

Changing what is collected can have privacy implications, so we have to ensure changes are properly reviewed. As an example of a privacy mistake, suppose a developer mistakenly decided it was important to understand which standard library packages are most imported and created a histogram of import paths using `counter.New("import:"+path).Inc()`. I don't think that would be a useful histogram anyway, but the privacy mistake is that the histogram would include private user import paths as well as standard library paths. However, the impact of the mistake would be limited to local collection, because the graphing and reporting configurations would not mention counters like `import:my.company/private/package`, so they would never be reported.

Developers of the Go toolchain will probably want to add counters to the toolchain purely for local use, to understand whether they would be helpful to report. That decision should not be overburdened with process, because the stakes are relatively low. Probably our standard code review process suffices, paired with clear documentation about what kinds of counters are and are not appropriate to introduce.

Changes to the server's graphing configuration merit more attention, since as we saw it is the graphing configuration that determines which counters are reported. It probably makes sense to require such changes to go through a review by a small group charged with ownership and maintenance of the configuration, either on the issue tracker or on the Gerrit server.

Finally, note the lack of any kind of wildcards in the graphing configuration. It is impossible to ask for all the counters beginning with `import:`, which means `import:my.company/private/package` will never be reported, because the graphing configuration will never list that counter explicitly by name. (Any attempt to do so would be caught by the public configuration review process.)

Reporting

When a counter file's week is over, toolchain programs (even long-running ones) automatically start writing counters to the next week's file. Remember that "week" refers to a 7-day period that starts on a weekday chosen randomly for each Go installation: on some machines weeks are Sunday to Saturday, others use Tuesday to Monday, and so on. At some point after the week ends, a reporting program (probably the `go` command, perhaps also `gopls`) will notice the completed week of counters and begin the reporting process.

The reporting program uses a reporting configuration to find out which counters should be reported. It would be served as a Go module (perhaps `telemetry.go.dev/config`). Visiting that same page in a browser would print a nice HTML page listing all the counters that have ever been collected, annotating each with the date ranges when it was collected and the justification for collection. In the event that some counter is deemed no longer necessary or somehow problematic to collect, it can be removed from the configuration, and programs will immediately stop reporting it. Similarly, if the system must be shut down for some reason, serving an empty configuration would stop all reporting.

The reporting configuration would be JSON corresponding to the Go type `ReportConfig` defined as:

```
type ReportConfig struct {
    GOOS      []string
    GOARCH    []string
    GoVersion []string
    Programs  []ProgramConfig
}

type ProgramConfig struct {
    Name     string
    Versions []string
    Counters []CounterConfig
    Stacks   []CounterConfig
}

type CounterConfig struct {
    Name string
    Rate float64
}
```

The `ReportConfig` lists the known GOOS, GOARCH, and Go versions that can be reported. This ensures that programs testing with an experimental, as-yet-unknown operating system, architecture, or Go version are not accidentally collected. Similarly, the `ProgramConfig` lists the programs that should be collected from and their specific versions, if they are separate from the main Go toolchain (like `gopls` and `govulncheck`). The `CounterConfig` lists the specific counters being collected and their individual sample rates.

The reporter starts by picking a random floating point number X between 0 and 1. If $X \geq 0.1$, then the reporter stops without even downloading the configuration. For example, if the reporter picks $X = 0.2$, it stops immediately. This step imposes a hard limit of 10% sampling rate for any counter or stack, and it arranges that a particular Go installation won't even download the collection configuration more than once every couple of months on average.

Assuming $X < 0.1$, the reporter downloads and reads the collection configuration. It then reads all the per-program counter files and filters them to include only the ones with matching GOOS, GOARCH, Go version, program name, and program version. It further filters the selected reports to drop any counters for which the configured rate is less than X. For example, if the reporter picks $X = 0.05$, it will report counters configured with rate 0.1 but not counters configured with rate 0.01. If a particular program has no sampled counters, that program is dropped from the report. If the report has no programs, no report is sent at all.

In a large deployment such as Go's, a typical reporting rate will be under 0.02 (2%), with the effect that each system will average around one weekly report per year, or fewer. One nice property of transparent telemetry is that as more and more systems run with it enabled, each system reports less and less data.

[*Update*, 2023-02-24: The hard limit of 10% and the expected reporting rate of 2% were based on opt-out telemetry with millions of installations. The design has changed to be opt-in*, which will raise those probabilities.]

When there is a report to send, the reporting program prepares JSON corresponding to the Go type `Report` defined as:

```
type Report struct {
    Config   string
    Week     string
    LastWeek string
    X        float64
    Programs []Program
}

type ProgramReport struct {
    Program   string
    Version   string
    GoVersion string
    GOOS      string
    GOARCH    string
    Counters  []Counter
    Stacks    []Counter
}

type Counter struct {
    Name  string
    Count int64
    Stack []string
}
```

The `Report`'s `Config` field lists the configuration version used for generating the report, so analysis can determine the sampling rates applied.

On a system that uses Go only intermittently, a reporting program might not run for a few days or more after the week ends. The `Report`'s `Week` field identifies the week this report covers, by giving its first day in `yyyy-mm-dd` format. If it has been more than seven days since the last use of Go, the now-weeks-old local report will not be uploaded. This lets the server "close the books" on a given week's telemetry after seven days.

In any data collection system it is important to quantify how much data is being discarded. (This is why, for example, `pprof` attributes missed profile events to synthesized functions like `_LostExternalCode`.) In transparent telemetry, if a system is used one week but then not used at all the next week, the system will have no opportunity to (randomly decide to) report the first week's data. The number of systems being used so intermittently is probably low enough not to worry about having a statistically significant effect on the results, but it would be good to measure that rather than guess. The `LastWeek` field reports the week prior to the one being reported when the reporting system last gathered any counters at all. On a frequently used system, `LastWeek` will always be seven days earlier than `Week`. After a long pause in Go usage, `LastWeek` will be two or more weeks earlier than `Week`, indicating that this system never even considered reporting counters from `LastWeek`. If a substantial number of reports have a mutiweek gap, we can conclude that the earlier week's data may be less accurate than previously estimated. Again, this is generally unlikely, but perhaps it would happen after vacations such as end-of-year holidays. It would be good to have an explicit signal that those numbers are not as trustworthy rather than puzzle through why they look different. The `LastWeek` field also makes it possible to estimate the number of active users over longer time periods, such as 4 weeks or 52 weeks, which may be useful for understanding overall usage.

Note that the different programs' counter sets are all uploaded together, so

that for example if the `go` command is taking a surprisingly long time to run a build, the associated counters from the `compile` and `link` program are in the same record. Note also that there is no persistent identifier in the records that would allow linking one week's upload with a different week's upload.

The server would necessarily observe the source IP address in the TCP session uploading the report, but the server would not record that address with the data, a fact that can be confirmed by inspecting the reporting server source code (the server would be open source like the rest of Go) or by reference to a stated privacy policy like the one for the Go module mirror*, depending on whether you lean more toward trusting software engineers or lawyers. A company could also run their own HTTP proxy to shield individual system's IP addresses and arrange for employee systems to set `GOTELEMETRY` to the address of that proxy. It may also make sense to allow Go module proxies to proxy uploads, so that the existing `GOPROXY` setting also works for redirecting the upload and shielding the system's IP address.

Recall from above that the local, binary counter files are stored in `<user>/go/telemetry/local/`. When a report is uploaded, the exact JSON that was uploaded is written to `<user>/go/telemetry/uploaded/`, named for the day of the upload (`2006-01-02.json`). The aim of both these directories (including their naming) is to make the system's overall operation as transparent as possible. The expectation is that a typical report will be under 1,000 counters, requiring about 50 kB in JSON format. Assuming twice as many counters are counted locally than are uploaded, that's 2,000 counters in binary format, which is another 100 kB. The storage cost of keeping the local forms indefinitely is then under 100 kB/week or 5 MB/year. An upload once or twice a year adds only another 100 kB/year. A command like `go clean -telemetry` would delete all of these.

The privacy feature of waiting at least a week before uploading anything at all (to give people plenty of time to opt out before any data is sent) means that ephemeral machines such as build containers will never be counted. The tradeoff of better privacy seems worth the loss of visiblity into these machines.

Publishing

Every day, the upload server takes the previous 24 hours' worth of uploads and updates the published graphs defined in the graph configuration.

It also publishes the full, raw JSON for the previous 24 hours worth of uploads, in seven distinct data sets corresponding to the seven different possible weeks (starting Sunday, Monday, Tuesday, ...) that could have been reported that day. For example, the files published on 2023-01-18 would be:

```
week-2023-01-04-uploaded-2023-01-17.v1.reports
week-2023-01-05-uploaded-2023-01-17.v1.reports
week-2023-01-06-uploaded-2023-01-17.v1.reports
week-2023-01-07-uploaded-2023-01-17.v1.reports
week-2023-01-08-uploaded-2023-01-17.v1.reports
week-2023-01-09-uploaded-2023-01-17.v1.reports
week-2023-01-10-uploaded-2023-01-17.v1.reports
```

Thanks to sampling, the collected uploads will be fairly small and will not grow even as the number of active installations does. Estimating 50 kB per uploaded report and a target of about 16,000 reports per week, each week's reports total only 800 MB (split across the seven different starting days in that week). Compression with Brotli should reduce the footprint by at least a factor of 10, making each week at most 80 MB, or at most 4 GB for an entire year's worth of uploads.

Opt-Out

[*Update*, 2023-02-24: The design has been changed to be opt-in*. This section is unmodified from the original for historical purposes.]

An explicit goal of this design is to build a system that is reasonable to have enabled by default, for two reasons. First, the vast majority of users do not change any default settings. In systems that have collection off by default, opt-in rates tend to be very low, skewing the results toward power users who understand the system well. Second, the existence of an opt-in checkbox is in my opinion too often used as justification for collecting far more data than is necessary. Aiming for an opt-out system with as few reasons as possible to opt out led to this minimal design instead. Also, because the design collects a fixed number of samples, more systems being opted in means collecting less from any given system, reducing the privacy impact to each individual system.

Enabling the system by default requires proper notice to users who are installing the system. As we did with the on-by-default module proxy and checksum database, notices would be posted in the release notes for the first Go distribution that enables telemetry as well as displayed next to the download links on `go.dev`* and `go.dev/dl`*.

Some users will want to opt out on general principle, no matter how minimal the system is, and that should be as easy as possible, something like:

```
go env -w GOTELEMETRY=off
```

Like all `go env -w` commands, this would configure a per-user setting that applies to all installed Go toolchains, present and future: a new Go toolchain installed tomorrow would respect the setting too.

In addition, some Linux distributions may want to prompt users during installation or disable telemetry unconditionally. We should make that easy to do too. Proposal #57179* introduced a `go.env` file in the root of the Go toolchain that configures per-toolchain settings. This will ship in Go 1.21. Linux distributions that want to disable telemetry could include a `go.env` file containing `GOTELEMETRY=off`.

Another dark pattern in opt-out systems is reporting information before the user has a chance to opt out. For example, I was once told about a popular developer tool that showed a telemetry checkbox, pre-checked, during the installation process, giving users the opportunity to uncheck the box. But at this point, a few screens into the installation, telemetry had already been sent, allowing the company behind the tool to track installation counts and opt-out rate by the fact that telemetry suddenly stopped, as well as tracking details like the IP and MAC addresses of systems that have opted out. In that system, to avoid sending any telemetry at all, you had to set an environment variable and then invoke the installer from the command line. I can't find concrete evidence anywhere for this story, so I am not sure if the system in question still behaves this way or ever did. Either way, I strongly disagree with this kind of trick as violating the entire spirit of an opt-out decision.

Transparent telemetry waits at least a week after installation before sending any report or even fetching the collection configuration. This should give plenty of time to run `go env -w` to opt out.

Summary

Repeating the summary from the introductory post*, transparent telemetry has the following key properties:

- The decisions about what metrics to collect are made in an open, public process.
- The collection configuration is automatically generated from the actively tracked metrics: no data is collected that isn't needed for the metrics.
- The collection configuration is served using a tamper-evident transparent log, making it very difficult to serve different collection configurations to different systems.
- The collection configuration is a cacheable, proxied Go module, so any privacy-enhancing local Go proxy already in use for ordinary modules will automatically be used for collection configuration. To further ameliorate concerns about tracking systems by the downloading of the collection configuration, each installation only bothers downloading the configuration each week with probability 10%, so that each installation only asks for the configuration about five times per year. [*Update*, 2023-02-24: The design has changed to be opt-in*, which requires raising these probabilities.]
- Uploaded reports only include total event counts over a full week, not any kind of time-ordered event trace.
- Uploaded reports do not include user IDs, machine IDs, or any other kind of ID.
- Uploaded reports only contain strings that are already known to the collection server: counter names, program names, and version strings repeated from the collection configuration, along with the names of functions in specific, unmodified Go toolchain programs for stack traces. The only types of non-string data in the reports are event counts, dates, and line numbers.
- IP addresses exposed by the HTTP session that uploads the report are not recorded with the reports.
- Thanks to sampling*, only a constant number of uploaded reports are needed to achieve a specific accuracy target, no matter how many installations exist. Specifically, only about 16,000 reports are needed for 1% accuracy at a 99% confidence level. This means that *as new systems are added to the system, each system reports less often*. With a conservative estimate of two million Go installations, about 16,000 reporting each week corresponds to an overall reporting rate of well under 2% per week, meaning each installation would upload a report on average less than once per year. [*Update*, 2023-02-24: The design has changed to be opt-in*, which requires raising these probabilities.]
- The aggregate computed metrics are made public in graphical and tabular form.
- The full raw data as collected is made public, so that project maintainers have no proprietary advantage or insights in their role as the direct data collector.

– The system is on by default, but opting out is easy, effective, and persistent. [*Update*, 2023-02-24: The design has been changed to be opt-in*.]

Next Steps

For more background about telemetry and why it is important, see the introductory post*. For more use cases, see the next post*.

Although these posts use Go as the example system using transparent telemetry, I hope that the ideas apply and can be adopted by other open-source projects too, in their own, separate collection systems.

I am posting these to start a discussion about how the Go toolchain can adopt telemetry* in some form to help the Go toolchain developers make better decisions about the development and maintenance of Go. I have written an implementation of local counter collection to convince myself it could be made cheap enough, but no other part of the system exists today in any form. I hope that the system can be built over the course of 2023.

* Asterisks mark hyperlinked text.