Parameterized Sampling Specification

2019-08-13

This is a working specification for release to the FoSS distributed tracing community

Curtis Bates

curtis.l.bates@gmail.com

Table of Contents

Summary

Although tracing exposes the inter-connectivity of our distributed systems, it is feared that too high of a sampling rate can drown system performance. In addition, sampling requests may come from outside the system boundary requesting sampling to continue (in the enterprise case). Parameterized or tunable sampling hopes to address these concerns.

Overview

Distributed Tracing is the act of sampling threads of execution throughout implemented business processes across one or more discrete processes. Distributed tracing is a valuable technique for looking into the performance of our business processes through empirical analysis. It's good to make data-informed decisions!

Let's consider a few cases that are fleshed out below in the Use Cases section. 1) High-Volume Tracing and 2) Targeted Tracing.

1) High-Volume Tracing

As a trace is started on a given thread of execution, it winds through connected systems as the business process is fulfilled. Many times these threads start with or quickly encounter composite services which require atomic services for their completion. This cascading service call behavior is nothing new to SOA or Microservice architectures and is the reason for distributed tracing. As it turns out, though, composite services tend to use the same atomic services - hence a feature we call re-use comes into play. Higher volume on these reused services means that they will statistically sample more often.

High-Volume tracing, also known as "auditing" is when a system performs "alwaysSample." The system sends a sampling request with every call downstream. Depending on your scenario, this may be worth it, but it defeats the point of being able to trend a system's performance with marginal overhead. With no regard to what the actual business dependencies are between these systems, an enterprise may end up overwhelming core services! For example, if all systems trace at 10% of requests yet all use a common CAS (Corporate Authentication Service), the CAS sampling requests will grow by the factor of number of calling services. This is nothing new - a centralized service must handle more load and the same will apply to sampling demand. The business should scale the services to handle additional business load, however, not additional tracing load. What if the business could throttle, back-off or black-list certain services that not only request sampling, but do so at high rates?

2) Selective Tracing

Selective tracing takes a rather different approach. A business can decide to trace a given thread of execution when the metadata conditions warrant a trace. For example, a particular user's interaction just never seems to work with a certain service. We could enable tracing for that user's cookie ID or session or X.509 credential or source IP, etc. In fact, the business help-desk could leverage this feature to have the customer "try again", with targeted sampling at 100% or alwaysSample, yet avoid the risk of imposing undo stress on the system at-large.

Both of these cases have static and dynamic natures to them.

Use-Cases

#1 Throttling of High-Volume, Upstream Sampling Requests

Educated speculation leads members of the distributed tracing community to realize that over-sampling in a large, distributed-tracing enabled system will lead to bottlenecks in service performance should downstream services (and / or systems for that matter) become "strange-attractors" - highly-connected, common services in the greater enterprise architecture. In order to back-off some of this impending load, a capability is needed to throttle inbound tracing requests regardless of the upstream system's desire to perform a trace.

Actors

Name	Description	Role (Primary / Secondary / External)	Goals (What are they trying to accomplish?)
Service	Receiving service in kind:SERVER position	Primary	Throttle sampling request
Client	Calling client from Upstream System in kind:CLIENT position	Secondary	Sample current thread of execution
Rules Engine	Component of the Service used to match Client to desired sampling rate	Secondary	Resolve Client to a matching substitute sampling rate
Collector Service	This is the service receiving emitted span data	External	Correlate spans into complete trace graphs

Preconditions

Calling client passes the following headers with sampling requested:

Header	value	Example

traceparent	<version>-<valid traceId>-<valid parentSpanId>-<traceOptions 01xH>	`traceparent: 00-462f5df1d8c7265b663cbb02e1234c7b-6865db71bcb41c74-01`
tracestate (emerging)	Vendor-specific rendering of trace information	

A mechanism for matching Client criteria exists in the Server such that:

1. The Server can distinguish the Client from other Clients
2. The Server is presented an alternative sampling rate for the Client for the given request
3. The new sampling rate overrides the default sampling rate at the time of Sampler creation
4. The new sampling rate impacts "shouldSample" decisions resulting in a sampling rate desirable for the Service (equal to or lower than the ephemeral rate imposed by the Client).

Let's call this mechanism the Rules Engine.

Client is initiating sampling at a high rate. For demonstration, Client is sampling with "alwaysSample" set. Every request contains a sampling request (traceOptions=01)

Triggers

Client contains criteria which matches information presented by the upstream down-sampling rule criteria held by the Server.

Post-Conditions

Service does NOT sample for this Client request based on the request of the Client, but on a fraction there-of.

A consequence of down-sampling is loss of granularity in completed traces such that the Client may have CLIENT span end without the Service contributing any more spans to the trace.

Sunny-Day Path

Both Client and Server vendor or trace to the same trace collector service . Because of this, the Client passing "sampling" requests to the Service to sample are coherent. Service, however, experiences high request volume from this Client and Client has "alwaysSample" set. The systematic error is the misunderstanding that sampling is "audit." To combat this behavior, Service down-samples Client sampling requests.

Alternative Path #1 - Service Follow-On Sampling Request

Service does not export spans for this Client request, but instead creates a new trace as though brand new. Although possible to decide firstly not to sample based on an upstream match

Alternative Path #2 - Client calls Service with different tracestate Header Set

Why is this Client asking this Service to sample when the emitted spans will not report to the same span collector (have different tracestates)? This is a futile request for sampling. The Service should reject the request to sample based on the fact the tracestates do not carry the same vendor Id. Any decision to sample is strictly the Service's as though it were initiating a new trace.

#2 Selective Sampling

Hard-to-troubleshoot situations arise in any system of reasonable complexity. In such a case, it may be advantageous to "trace" calls particular to a client's attributes - a cookie Id, hostname, etc. perhaps tracing of endpoints marked as "dev" or "beta" are desirable. Distributed tracing techniques are crucial to enhancing our observability of the system at large. Deployment decisions of new features or troubleshooting of existing ones requires knowledge of their impact to the business flow, not just knowledge of their individual atomic, functional quality - something we glean from unit testing, etc. Services, depend on one another in a distributed architecture, having systemic impact.

Actors

Name	Description	Role (Primary / Secondary / External)	Goals (What are they trying to accomplish?)
Service	Receiving service in kind:SERVER position	Secondary	Inject metadata into the sampler creation method
Client	Calling client from Upstream System or browser	Secondary	User of the service above
Rules Engine	Component of the Service used to match Client to desired sampling rate	Secondary	Resolve Client to a matching substitute sampling rate
Sampler	The logic that will make a decision to sample based on environmental criteria	Primary	Determine whether to sample based on environmental criteria

Preconditions

Calling client passes the following header(s) with sampling requested:

Header	value	Example
header	A key=value pair of metadata	user-agent=`python-requests/2.18.4`

Rules exist such that a match can be made against such a header:

Syntax	Example Rule	Corresponding Sampling Rate	Example Matching Header
RQL	user-agent=like=python*	80%	user-agent=`python-requests/2.18.4`, user-agent=python-requests/3.1.1
RQL	and(user-agent=like=python*, username=testuser)	100%	user-agent=python-requests/3.1.1, user=testuser

RQL (Resource Query Language) syntax is particularly interesting due serialization characteristics and AST (Abstract Search Tree) conversion.

Triggers

A new request is created against a Service endpoint.

Post-Conditions

Service invokes creation of a sampler that is created with knowledge of key-value pairs from the environment.

Sunny-Day Path

When the Client connects to the Server, the Server must decide whether to create a new trace and sample. Using environmental metadata from the client and rules in the Rules Engine, matches to metadata are made. This information is passed to the Sampler to inform the decision to sample. The business logic for this Sampler decides to sample if:

1. Trace doesn't already exist
2. Sampling is not disabled
3. Matches exist
4. The Matches are from selective Tracing rules

The corresponding sampling rate for the matching rule is used to finally determine the decision to sample.

#3 Shunt Sampling

In the case where sampling becomes problematic, the need may arise to simply shut it off. The SA or system monitor (human or automated) needs a control-path to disable sampling.

Actors

Name	Description	Role (Primary / Secondary / External)	Goals (What are they trying to accomplish?)
Service	Receiving service in kind:SERVER position	Secondary	Sample the new Service request from the Client
Client	Calling client from Upstream System in kind:CLIENT position	Secondary	Initiate a Service request
Rules Engine	Component of the Service used to match Client to desired sampling rate	Primary	Provide external / out-of-band control

Preconditions

Rules Engine has state concepts that carry from thread-to-thread, with process scope. This may be implemented as a Singleton, much like the OpenCensus Tracer.

Rules Engine receives input to disable tracing.

Triggers

A new request is created against a Service endpoint by the Client.

Post-Conditions

The Service does NOT create a trace and / or operates a trace using the NooP or NeverSampler

Sunny-Day Path

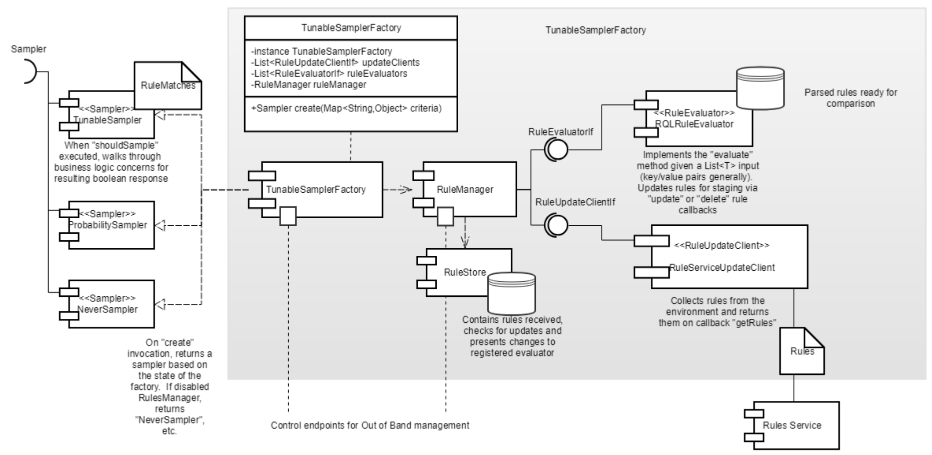
When the Service receives an endpoint invocation and begins to make the decision to trace, the decision is shunted by the state of the Rules Engine and / or the recent configuration of the Tracer. This causes the Service to continue with a SpanContext and associated Sampler that will not sample. No additional code is required for the Service to determine whether-or-not to sample as this is encapsulated in the Rules Engine / and or Tracing apparatus. The thread of execution continues invoking sampling from method to method as decorated, yet none of these actions results in any sampling and never any sampling data being exported.

Conclusion

Now that we've seen cases where throttling and / or selection may be needed, we must consider how this occurs. This is the purpose for this spec.

Proposed Implementation

A proposed solution is a dynamic sampler factory and rules management ecosystem - a way to create samplers based on the condition of a given request.


Sampler
RuleMatches
<<Sampler>>
TunableSampler
When "shouldSample" executed, walks through business logic concerns for resulting boolean response
<<Sampler>>
ProbabilitySampler
<<Sampler>>
NeverSampler
On "create" invocation, returns a sampler based on the state of the factory. If disabled RulesManager, returns "NeverSampler", etc.
TunableSamplerFactory
-instance TunableSamplerFactory
-List<RuleUpdateClientIf> updateClients
-List<RuleEvaluatorIf> ruleEvaluators
-RuleManager ruleManager
+Sampler create(Map<String,Object> criteria)
TunableSamplerFactory
TunableSamplerFactory
RuleManager
RuleStore
Contains rules received, checks for updates and presents changes to registered evaluator
Control endpoints for Out of Band management
RuleEvaluatorIf
RuleUpdateClientIf
<<RuleEvaluator>>
RQLRuleEvaluator
Parsed rules ready for comparison
Implements the "evaluate" method given a List<T> input (key/value pairs generally). Updates rules for staging via "update" or "delete" rule callbacks
<<RuleUpdateClient>>
RuleServiceUpdateClient
Collects rules from the environment and returns them on callback "getRules"
Rules
Rules Service

The components of this ecosystem are as follows:

1. TunableSamplerFactory
2. RuleManager
3. RuleStore
4. RuleEvaluator(s)
5. RuleUpdateClient(s)
6. RuleService
7. TunableSampler

Models presenting themselves in this design are:

1. Rules
2. MatchRules
3. RuleMatchResults

TunableSamplerFactory

Responsibilities

The TunableSamplerFactory creates Samplers given rule matching criteria (key/value pairs), returning Samplers that extend the Abstract Sampler.

Creation

The TunableSamplerFactory is created as a Singleton at the beginning of the application.

The TunableSamplerFactory instantiates any other infrastructure necessary to function:

- RulesManager (and RulesStore)

Concerns

Concern 1 - Sampler Creation

The TunableSamplerFactory.instance().create(criteria) uses this criteria to setup the new TunableSampler and chooses a Sampler to return based on the outcome of the RuleEvaluator. Possible outcomes are:

1. RuleManager has sampling disabled
2. Rules Match
3. No Rules Match

If no rules match, the "ProbabilitySampler" is returned with the default sampling rate set.

If the RulesManager has been disabled, then the "NeverSampler" returns.

If any of the remaining conditions are true, the "TunableSampler" is returned.

TunableSampler

Responsibilities

The TunableSampler makes the final "shouldSample" decision (TRUE/FALSE) given the following criteria:

1. Matching Rules (whether UpstreamSystem filter or Selective filter)
2. TraceContext (Parent, remoteParent or no parent)
3. Sampling state (Enabled / Disabled)

The TunableSampler may sort MatchedRules and process them by type, in order (ascending), lowest first giving rise to different responses given the condition.

The TunableSampler returns a rational description and /or matching rule as the reason for sampling.

Creation

The TunableSampler is created by the TunableSamplerFactory and extends the Abstract Sampler.

Concerns

Concern 1 - Exposure to Content

When the Tracer invokes "shouldSample()" on the TunableSampler, it leverages the current MatchedRules results held in the TunableSampler, as well as other knowledge about the environment as part of the input from the shouldSample method:

```
        @Override
        public final boolean shouldSample(
                        @Nullable SpanContext parentContext,
                        @Nullable Boolean hasRemoteParent,
                        TraceId traceId,
                        SpanId spanId,
                        String name,
                        @Nullable List<Span> parentLinks) {
```

Concern 2 - Encapsulation of Sampling Business Logic

The TunableSampler can perform the rest of the business logic processing, this time with request context already having been established:

1. If a traceId / parentContext already exists,
    a. if parentContext.samplingEnabled
        i. If the UpstreamSystemFilterRule matches throttling criteria, return a sampling decision base on
           the UpstreamSystemFilterRule's sampling rate and the random number set at this TunableSampler's creation
        ii. else, return TRUE (AlwaysSampler like logic)
    b. else
        i. if Client Criteria matches selectiveTracing rules
            1. return probabilitySampler using the matching selectiveTracing rule's sampling rate and the random number set at this TunableSampler's creation
        ii. else
            1. Return default a decision based on the defaultSamplingRate and the random number set at this TunableSampler's creation
2. else, (This is a new trace)
    a. if Client Criteria matches selectiveTracing rules
        i. return probabilitySampler with matching selectiveTracing rule's sampling rate and the random number set at this TunableSampler's creation
    b. else
        i. Return default a decision based on the defaultSamplingRate and the random number set at this TunableSampler's creation

RulesManager

Responsibilities

The RulesManager maintains state of the current list of rules (RulesStore).

The RulesManager proxies requests for rule evaluation, which is performed by a RuleEvaluator

The RulesManager presents rule(s) update to its internal state of rules (RulesStore) and RuleEvaluator when new or updated

The RulesManager starts and pulls rules from registered RuleUpdateClient(s) at the configured periodicity

Creation

RulesManager is created by the TunableSamplerFactory.

The RulesManager receives updates from registered RulesClient(s) that read from a Rules Service or local file, etc. following a schema for encoded rules and their sampling rates. These rules are ready for reference with every invocation of the TunableSamplerFactory.create(.criteria).

When the invocation occurs, the RulesService takes the incoming criteria and provides it to a RulesEvaluator. The resulting MatchedRules are provided to the TunableSampler for the final sampling decision.

The workflow has 2 primary concerns as follows:

Concerns

Concern 1 - Search Criteria

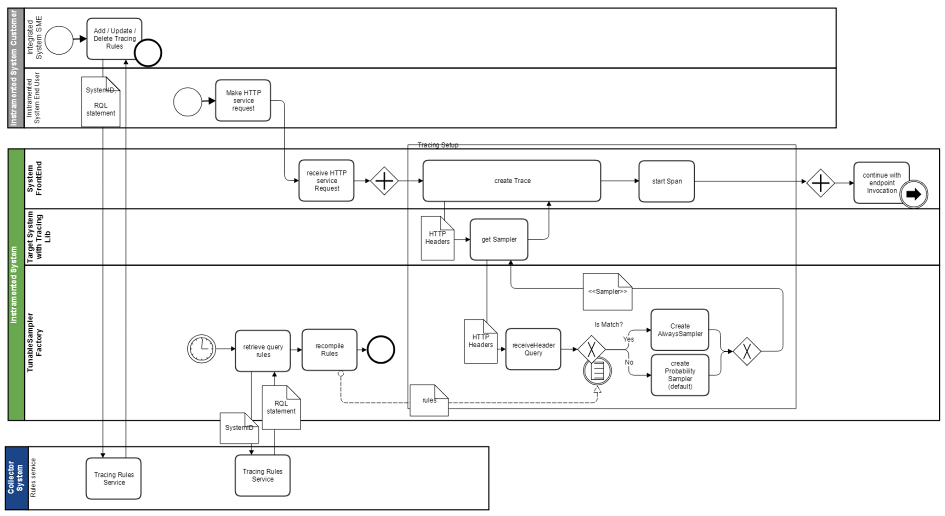
The primary workflow involves supplying environmental criteria to the TunableSamplerFactory.instance().create(criteria...) to pass along to the RulesService to retrieve matching matching rules at the time of Sampler creation. A RulesService provides a separation-of-concerns for rules from the TunableSamplerFactory, yet also delegates rule evaluation to a RuleEvaluator.

Concern 2 - Update Rules

The second workflow involves allowing the Integrated System SME the ability to set custom rules for their tracing to selectively trace on.

The rules for matching are retrieved from the Rules Service or file by a RulesUpdateClient at a prescribed update interval. The update of rules does not impact the operation of the application, but instead become available at the next sampler factory creation event. The update of rules is orthogonal to normal business operation of the Service and should never render a blocking event of any business service.

When a rule is received, the RulesService responsibility is to determine whether the rule is new or an update, or a duplicate. If the rule is new or updated, it is presented to the registered RuleEvaluator for preparation. If the rule is a duplicate, it is discarded.



RuleEvaluator

Responsibilities

The RuleEvaluator is opinionated, built to process rules of a given dialect (for example, RQL, SQL, Simple key/value matching, Regex)

The RuleEvaluator must handle a Map or List of key-value pair input.

The RuleEvaluator must return a List of matching rules.

Creation

A RuleEvaluator of the desired type is created and registered with the RulesManager. The RuleEvaluator must implement the interface RuleEvaluatorIf.

Concerns

Concern 1 - Rule Update

Rule updates are presented to the RuleEvaluator by the RulesManager. The RuleEvaluator recompiles or prepares the rule and stores it for use, discarding any previous reference to a matching rule.

Concern 2 - Rule Preparation

As has been seen with RQL or Regex, there's a difference in the rule syntax and its compilation. RuleEvaluators receive ruleUpdates allowing the RuleEvaluator to prepare a compiled rules reference ready for execution against search requests.

Concern 3 - Rule Query

Queries are performed by the RuleEvaluator. The RuleEvaluator must be able to search across presented criteria (Map<String, Object>) for example and return a List<MatchRules> which may be empty. The sophistication and implementation of the search function is encapsulated in the RuleEvaluator.

RuleUpdateClient

Responsibilities

The RuleUpdateClient is responsible for retrieving rules from a rules source, such as a local file or rules service.

The RuleUpdateClient should validate rules

The RuleUpdateClient should marshal rules into the Rule data model

The RuleUpdateClient should fail gracefully if issues arise with a given update without impacting the integrity of the RulesManager.

The RuleUpdateClient should retrieve rules on callback by the RulesMananger

Creation

The RuleUpdateClient is created and registered with the RulesManager. The RuleUpdateClient must implement the RuleUpdateClientIf.

Concerns

Concern 1 - Rule Retrieval

The RuleUpdateClient is concerned with retrieving and presenting rules to the RulesManager in the known Rule Class model. The following is an example Schema and example of Rules:

JSON Rules Schema:

```
{
  "version": {
    "type": "string",
    "required": true
  },
  "dialect": {
    "type": "string",
    "required": true
  },
  "resourceId": {
    "type": "string",
    "required": true,
        "$comment": "The System ID for which these rules belong to"
  },
  "enabled": {
    "type": "boolean",
    "required": true,
        "$commment": "Sampling may be disabled via rule update"
  },
  "defaultSamplingRate": {
    "type": "number",
    "required": true,
    "$comment": "The default sampling rate."
  },
  "upstreamSystems": {
    "type": [
          {
        "query":   {"type": "string" },
            "sampling":{"type": "string" },
            "rate":    {"type": "number" },
        "enabled": {"type": "boolean"}
      }
    ],
    "required": false
  },
  "filterRules": {
    "type": [
          {
        "query":   {"type": "string" },
            "sampling":{"type": "string" },
            "rate":    {"type": "number" },
        "enabled": {"type": "boolean"}
      }
    ],
    "required": false
  }
}
```

Rules have an implied precedence due to their order. Rules should be processed in order and results returned with an order ordinal. Allowing the TunableSampler to sort the returned list and evaluate matches given the precedence.

Sample Rules:

```
{
   "version" : "1",
   "dialect": "RQL",
   "resourceId": "80eeab50-644b-11e8-b55a-xxxxxxx1",
   "enabled" : true,
   "defaultSamplingRate" : 1,
   "upstreamSystems" :
   [
      {
        "query" : "host=like=*.abcd.com",
        "sampling" :
        {
          "rate" : 100,
          "enabled" : true
        }
      }
   ],
   "filterRules" :
   [
     {
       "query" : "host=like=*.abcd.com",
       "sampling" :
       {
         "rate" : 100,
         "enabled" : true
       }
     }
   ]
 }
```

Solution Business Logic

The following is potential business logic that the TunableSampler may implement to decide whether or not to sample.

Cases are processed in-order as lower-order cases have precedence over higher cases. (case 1 before, case 2, etc.)

Case 1 - Sampling Disabled

Explanation:	If default sampling rate is set to 0.0d, then sampling is disabled. The default sampling rate field in the rules schema reflects the corporate sampling rate. The corporate sampling rate, for example may otherwise be 0.01d or 1/100% of the time a sample will be created.
Controlled by:	RulesManager
Initiator:	Set when instantiated at creation time and potentially collected by the RuleUpdateClient and upated at runtime.
Input:	Properties file, Rules
Conditions:	default sampling rate = 0.0d
Output:	No sampling. Sampling disabled.

Case 2 - Default Sampling due to Upstream Initiation

Explanation:	Upstream system calls target system with tracing headers set. The default behavior is to honor the trace request and produce a sample because of the presence of trace information. Traceparent includes traceOptions isSampled=1 concept. In this default case, sampling is performed with sampling rate at 100%.
Controlled by:	TunableSampler
Initiator:	Upstream (client)

Input:	"tracing information" in headers from upstream system
Conditions:	No matching UpstreamSystem filter rules
Output:	100.0% Sampling rate.

Case 3 - Sampling due to Upstream Initiation with Rate Filtering

Explanation:	By default, upstream systems cause sampling by sending trace information in their requests. Upstream filtering rules can throttle or reduce the trace sampling request rate, throttling upstream sampling should it be determined that an upstream system is swamping a downstream system with sampling requests. To throttle an upstream system, a throttle rule is put in place: resourceId=12345asdf would throttle a calling system presenting such an Id.
Controlled by:	FilterRules configured by system owner via Rules file or retrieved from Rules Service depending on the RuleUpdateClient used, RulesManager
Initiator:	Upstream (client)
Input:	"resourceId header", "tracing information"
Conditions:	Presented headers by upstream system contain the "resourceId header" for comparison against systems to throttle. UpStreamSystem(s) rules exist.
Output:	Sampling occurs at rate returned when matched (singular) to a upstreamSystem filter rule (rule has a rate assigned to it.) after sort. The rate returned is equal-to or less than the default Sampling Rate.

Case 4 - Sampling Initiated with FilterRules

Explanation:	When a client calls with no tracing information, the sampling client(server) may initiate trace if a filterRule matches any header or request key=value information.
Controlled by:	FilterRules configured by system owner via Rules file or retrieved from Rules Service depending on the RuleUpdateClient used, RulesManager
Initiator:	Upstream (client)
Input:	"header" / request session key/value pairs
Conditions:	default Probability != 0.0d, filterRule(s) exist and filterRule(s) match
Output:	Probability returned is that of matching rule (singular) after sort. Rule has sampling rate to return attached to it.

Case 5 - Sampling Initiated using Default Sampling Rate

Explanation:	Typical default probability sampling is performed when client calling contains no tracing request, yet our probability role determines that we should sample this request. The server creates a Trace and span due to this decision and passes downstream.
Controlled by:	ProbabilitySampler
Initiator:	Upstream (client)
Input:	Client call
Conditions:	default Probability != 0.0d, no upstream tracing request, no filterRules matched or no rules present.
Output:	Default sampling probability

Definitions

Definitions of terms used in this paper.

Term	Definition
trace	The grouping of all related spans in a graph representing a thread of execution through a business process. The thread may be synchronous or asynchronous or combination of both. There is no time restriction on the conclusion of a trace.
span	The measurement of time and status of a processing call, whether internal to a system a method call for example, or external, such as a client calling a server.
initiator	The actor making the decision to create a traceId and begin a span (the root span - where parentId=null).
sampling	The act of creating spans (that link to a trace)
reporter	The emitter of spans from a service to a collector
collector	The service receiving spans for correlation
traceparent	The Identifying information for propagating sampling information downstream
tracestate	The identifying information for propagating the target collector (and potentially system-namespacing within a large-scale enterprise collector)
UpstreamSystem	The system initiating or forwarding the sampling request to the service of concern.
FilterRules	The set of rules used to select from metadata provided when creating a TunableSampler. These rules may sample at a higher rate or a lower rate than what the default sampling rate is set at.
UpstreamSystem Rules	The set of rules used to select from metadata provided when creating a TunableSampler. These rules may sample at a higher rate or a lower rate than what the default sampling rate is set at. These rules look to filter given specific ids from upstream systems.
Sampling Rate	This refers to a percentage rather than a probablity - a.k.a, 10% ~ to the probability of 0.1