- Composite Probes – a Monitoring Framework for Organising Data into Configurable Hierarchies

- Ada Diaconescu -

Orange Labs - MAPS/AMS

Grenoble

February 2007

http://cvs.forge.objectweb.org/cgi-bin/viewcvs.cgi/lewys/CompositeProbes/

Table of Contents

1 Introduction

1.1 Motivation and Goals

System monitoring has become an essential utility for managing the development, configuration and runtime administration of software systems. Generic and specific monitoring tools are being employed to test and calibrate software systems offline and to supervise, manage and adapt software systems at runtime. Monitoring utilities are generally used to collect data of different types. Namely, collected data can range from a system's hardware and software resource consumption, to the system's usage patterns and achieved quality attributes. Monitoring data is subsequently analysed in order to determine a system's correctness, performance, overall client utilisation, or general SLA-conformance. Nonetheless, as systems are becoming increasingly complex and the available monitoring tools more sophisticated, progressively larger amounts of heterogeneous data are being collected for system analysis and diagnosis operations. Consequently, examining collected monitoring data is becoming an increasingly complex and costly task, especially when periodically required during system execution.

Currently, monitoring utilities generally represent collected data at the same abstraction level, in a flat data structure. This implies that measurements for abstract, higher-level resources are not readily available and must always be calculated based on the multiple low-level measurements available. For example, for calculating a cluster's CPU usage, one must repeatedly retrieve and aggregate data from the individual CPU probes available, corresponding to each processor present in that cluster. This process must be repeated separately by all parties interested in the CPU cluster measure.

Composite Probes provide support for organising and managing monitoring data in large-scale, distributed software systems. The goal is to extend flat monitoring architectures with hierarchical constructs, in order to improve understanding and analysis of massive amounts of monitoring data. Composite Probes are used as building blocks for constructing flexible monitoring hierarchies, which can process monitoring data at different granularity levels. Data obtained from low-level system probes is propagated upwards through the monitoring hierarchy and incrementally processed at each Composite Probe involved. Data-processing functions can be separately configured for each Composite Probe. Such configurable functions include data aggregation, filtering and scheduling for statistical calculation. The proposed framework is based on a modular, configurable and extensible architecture, which allows the provided functionalities to be individually tuned or modified. The framework's ongoing implementation is based on the Fractal component model.

1.2 The Composite Probes Monitoring Framework

Composite Probes is a generic monitoring framework for autonomic computing. It provides support for organising and managing monitoring data in large-scale, distributed software systems. The goal is to extend flat monitoring architectures with hierarchical constructs, in order to improve the understanding and analysis of massive amounts of monitoring data.

Composite Probes are used as building blocks for constructing flexible monitoring hierarchies, which can process monitoring data at different granularity levels. Data obtained from low-level system probes is propagated upwards through the monitoring hierarchy and incrementally processed at each Composite Probe involved. Data processing functions can be separately configured for each Composite Probe. The most important configurable functions include data aggregation, filtering and scheduling for statistical calculation. The framework is based on a modular, configurable and extensible architecture, which allows the provided functionalities to be individually tuned or modified. As a starting point, the current Composite Probes implementation uses and extends the monitoring part of the CLIF load-injection and monitoring tool, and is based on the Fractal component technology. Two main probe types are provided in the Composite Probes framework. Basic Probes are responsible for extracting monitoring data from the managed system resources. These probes cannot be used to manage and organise data from multiple data sources. The second probe type, Composite Probes, is provided for this purpose instead. Composite Probes are used to encapsulate data from multiple, finer-grained probes, in order to simulate abstract resources at a higher granularity level. For example, in the provided scenario in Figure 1, a Composite Probe is used to simulate a monitoring probe for a global cluster CPU resource.

1.3 Example Scenario

A possible scenario where the Composite Probes framework can be employed involves monitoring a computer cluster.

Figure 1 provides a simplified example of a clustered system with two interconnected machines. For monitoring this system, two Basic Probes are deployed on each machine for measuring their memory and CPU resource consumption. In a real scenario, it can be imagined that monitoring probes from a hundred machines would produce a large amount of low-level data, making it hard to manually analyse in real-time. To alleviate this difficulty, Composite Probes are used to aggregate monitoring data at different abstraction levels, such as system level and overall cluster level. In the example below, monitoring data extracted by the Basic Probes is aggregated as follows. Two Composite Probes are used to represent the overall load of each system (i.e. 'system1' and 'system2' Composite Probes). At a higher abstraction level, another Composite Probe is used to represent measurements of the overall cluster load (i.e. 'cluster' Composite Probe). Finally a Composite Probe is used aggregate CPU monitoring data from all cluster machines, so as to represent the overall cluster CPU load.

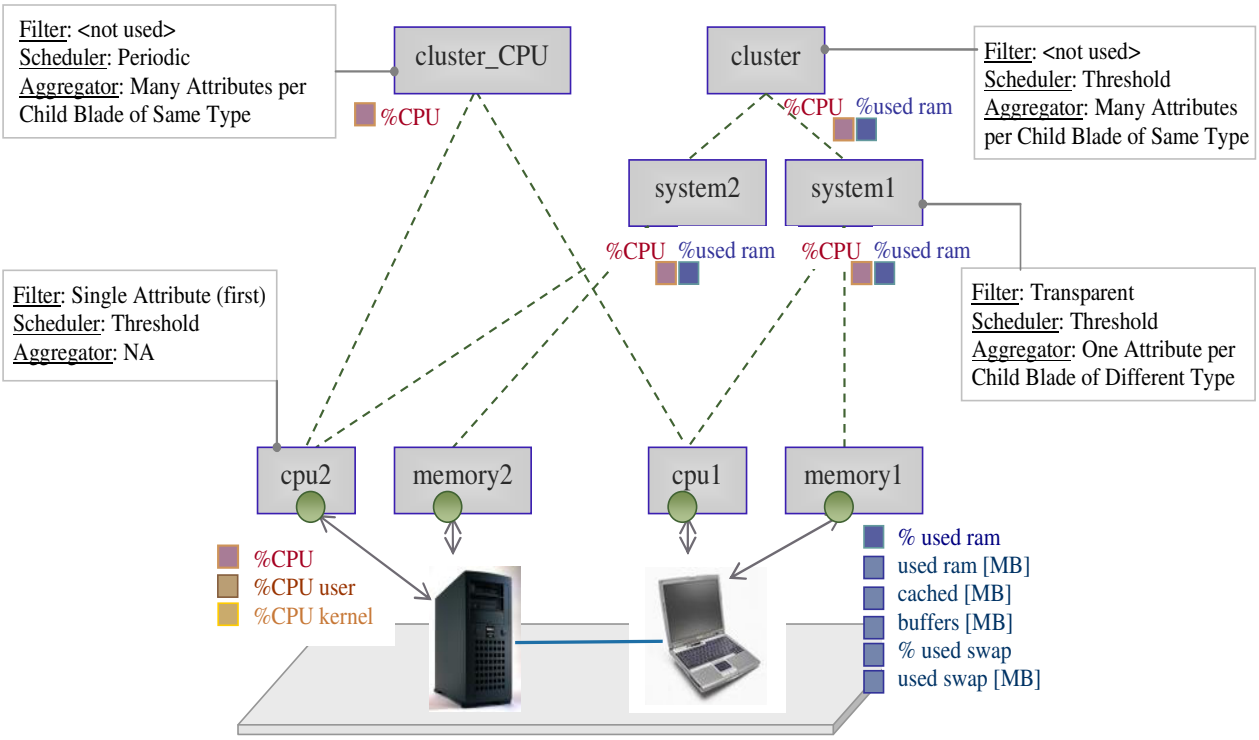

Figure 1: Example scenario of Composite Probes monitoring a clustered system

2 Composite Probes Overview

2.1 Composite Probe types

Composite Probes can be classified into two major types, based on the Probe's roles and functionalities (Figure 2). The first Probe type is the *Basic Probe*, whose role is to extract the actual monitoring data from the managed system resources. Basic Probes represent *leaf* nodes in the Probes hierarchy, meaning that they cannot be further composed of other Probes. Basic Probes locally process collected monitoring data and subsequently forward it to subscribed *parent* Probes.
The second Probe type is the *Composite Probe*, whose role is to manage and organize incoming monitoring data from multiple data sources, or *child* Probes. Composite Probes can contain other Composite and/or Basic Probes, which constitute the Composite Probe's data sources. Composite Probes locally process incoming monitoring data and subsequently forward it to subscribed *parent* Probes. Local data processing in Composite Probes involves data aggregation and filtering procedures.
From an external perspective, clients are unable to differentiate the available Probe types. All access to a Probe is restricted to the Probe's external interface(s), which is identical for all the Probe types (Figure 2).

A few probe-related naming conventions are in order at this point to help clarify the meaning of terms in this document. First, as a CLIF extension, *Probes* are sometimes also referred to as *Blades*. This is the more generic term used in CLIF to identify both monitoring probes and load injectors. A second convention, depending on the context, the term *Composite Probes* is sometimes used to refer to all Probe types in general, and sometimes to the specific Composite Probe type. This is because all Probe types may seem to be Composite Probes from an external client perspective. More precisely, clients are not aware of whether they enquire or control a Basic or Composite Probe. To avoid confusion, in unclear contexts the term Probe is simply used to generally refer to all Probe types. Finally, the term *Primitive Probe* is sometimes used to refer to the Basic Probe type.

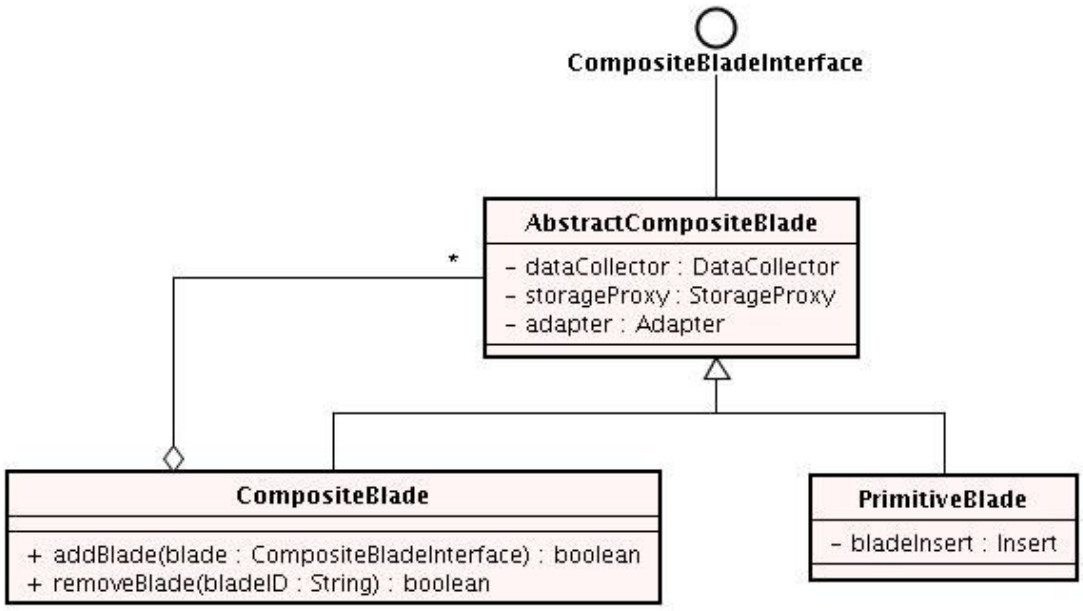

Figure 2: Composite Probe types:

***Primitive Probe* (or Blade) does *not* contain other Probes; it uses an Insert subcomponent to extract monitoring data directly from the system resources (also referred to as *Basic Probe*)**
***Composite Probe* (or Blade) contains other Composite or Primitive Probes**

2.2 Unique identification and routing

2.2.1 Unique identification

All Composite Probes can be uniquely identified via a Probe Id (or Blade Id). There are two possible approaches for specifying Probe Ids so as to ensure their global uniqueness through the monitoring hierarchy. The first approach is to label Probes with locally unique Ids and ensure the Probe's unique identification via its absolute location within the monitoring hierarchy. This approach is valid only in case the monitoring hierarchy is a of a tree type. This means that the hierarchy has a single root Probe and that each Probe in the hierarchy has a single parent Probe via which it can be reached. In this case, each Id must be unique only with respect to its parent Probe, meaning that all child Probes of a certain parent must have different Ids. In this scenario, each probe can be uniquely identified via the full path from the hierarchy's root to the Probe. In a tree-like hierarchy, a single path can exist between the tree's root and the targeted Probe. This property ensures each Probe's unique identification via this path. This approach has the advantage of not imposing absolute Ids over the entire monitoring hierarchy. This is an important benefit since the task of assigning absolute Ids can become difficult in large-scale monitoring networks. At the same time, this approach has the significant disadvantage of restricting the applicability context to tree-like hierarchies.
The second option is to label Probes that are globally unique within the monitoring hierarchy. This approach can support any graph-like hierarchy, where each Probe can be reached via several parent Probes. In addition, an external client trying to directly access a certain Probe is not required to be aware of the absolute placement of that Probe within the overall monitoring hierarchy. In the example in Figure 1, a client can directly access the System1 Composite Probe, in order to have monitoring data representing the system load. In case absolute Ids are used, the client will not have to be aware of the fact that the System1 Probe is actually a child of the Cluster Probe within a Probe hierarchy. Similarly in the example in Figure 5, a client can directly identify and access the targeted probe C3 via its unique Id, rather than specifying its path within the overall hierarchy. For these reasons, the second option was selected for the current Composite Probes implementation.
Other identification approaches can be envisaged based on different combinations of the two principal approaches above.

2.2.2 Routing

In the Composite Probes context, routing refers to directing information flows through the monitoring hierarchy from a source Probe to a targeted Probe. The rooting function is needed for two main purposes. One is to allow external clients require monitoring data from a targeted Probe, while making the request on a different Probe.
This option can be useful for allowing a client to request data from several lower level Probes, while holding a single reference to a high-level Probe in the hierarchy. In the example in Figure 1, a client can use the same cluster Composite Probe to request detailed monitoring data from all CPU and memory Basic Probes in the cluster.

Similarly, the second reason for the routing function is to allow clients to send control commands for a targeted Probe, while making the request on a different Probe.
In addition, it can also be considered that the routing function is needed to propagate monitoring data and control commands through the monitoring graph. In these scenarios, the routing information is needed to identify a Probe's child and parent Probes, so as to forward them the control commands and monitoring data, respectively.
A basic routing function is provided in the current Composite Probes implementation. The routing function is implemented in the Probe Delegation Manager subcomponent (subsection 3.5). The Delegation Manager uses routing tables to store routing information. In the current implementation routing data consists of the ids and locations of neighbouring Probes (i.e. child and parent Probes). In order to direct requests to Probes that are not in their immediate vicinity, Delegation Managers must currently receive the path from their Probe to the targeted Probe. In the example in Figure 1: Example scenario of Composite Probes monitoring a clustered system a client request received by the cluster Probe and targeted to the CPU1 Probe will have to provide the "cluster/System1/CPU1" path as a parameter. Only the path between the receiving node and the targeted one is required, rather than the full path from a root node to the targeted one.
The current identification and routing solution was chosen for its provided simplicity while meeting all current requirements in an arbitrary graph hierarchy. The solution can be extended to provide Delegation Managers with further routing information and capabilities, and consequently avoid requiring path information from every request.

2.3 Data types

This section discusses:
⇨ Data types managed by Composite Probes:
- long[] monitored parameter values
- String[] monitored parameter names, or labels

⇨ Performance considerations supporting this choice
⇨ Future extensions (section 6.1)

2.4 Information flow through Composite Probes

Two main information flows characterise monitoring hierarchies based on Composite Probes. These are the data flow and the control flow. In short, the data flow transports monitoring data from the lower-level Basic Probes up through the hierarchy to the root-level Composite Probes. The control flow transports control commands from Composite Probes at higher hierarchical levels down through the hierarchy to the Basic Probes. The two information flows are discussed in more detail over the following subsections (2.4.1 and 2.4.2).

2.4.1 Data flow

The data flow transports monitoring information through the Composite Probes hierarchy. Monitoring data is propagated upwards from the Basic Probes collecting the data to the Composite Probes at the top of the hierarchy. Figure 1: Example scenario of Composite Probes monitoring a clustered system depicts the general data flow through a monitoring Probe and introduces the main related concepts and terms. In short, a Probe will generally receive monitoring data from one or multiple *data*

sources. Incoming data will be locally processed and stored as *local data*. Local data represents the data monitored by the probe. For Basic Probes, local data is the data the probe has actually monitored. In the case of Composite Probes, the local data simulates monitored data as if measured from a higher-level resource that the probe represents. Local data is subsequently forwarded to *data sinks*, possibly being processed or filtered before. Data sinks generally consist of Composite Probes at upper hierarchical levels, representing monitoring probes for simulated resources at higher abstraction levels. In addition, external clients can directly enquire a Probe for retrieving its local data.

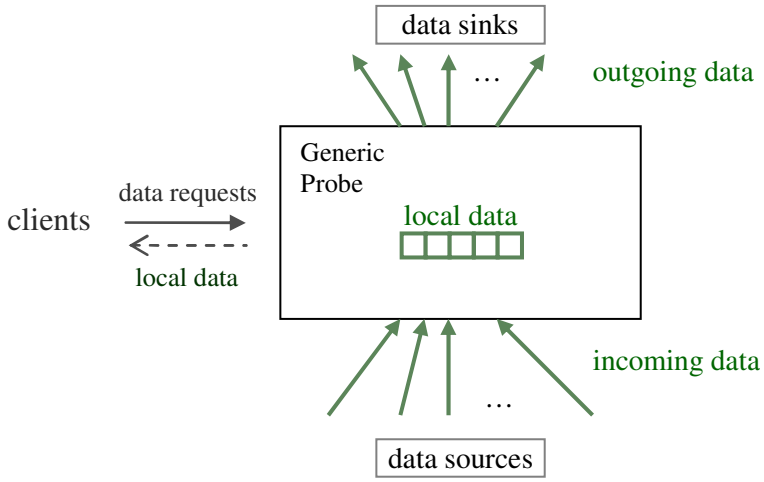

Figure 3: Data flow through Composite Probes
The probe receives data from multiple sources, aggregates the data and stores it locally
The probe filters the local data and sends the result to multiple data sinks
The probe's clients make direct requests for the local data

Figure 4 details the data flow view for Basic and Composite Probes. It indicates the main data-related differences between the two Probe types. The main dissimilarity is in the source of monitoring data in the two cases. That is, Basic Probes receive data from a single data source, which is the instrumented system resource they monitor. On the other hand, Composite Probes receive data from multiple sources, and must be able therefore to aggregate them into meaningful local data. Basic Probes do not need this functionality since their monitoring data can be directly stored as local data. Local data events can be used to calculate summary statistics in both Basic and Composite Probes. Statistics are used to summarize a set of observations, in order to communicate as much as possible and as simply as possible to external clients and higher-level Composite Probes. Such functions can include various mean or median functions, standard deviation, variance, minimum or maximum functions. In addition, both Probe types can filter their local data and forward it to multiple data sinks, or Composite Probes. The two Probe types are transparent from a client's perspective, as clients see all probes as Composite Probes, from a functional point of view.

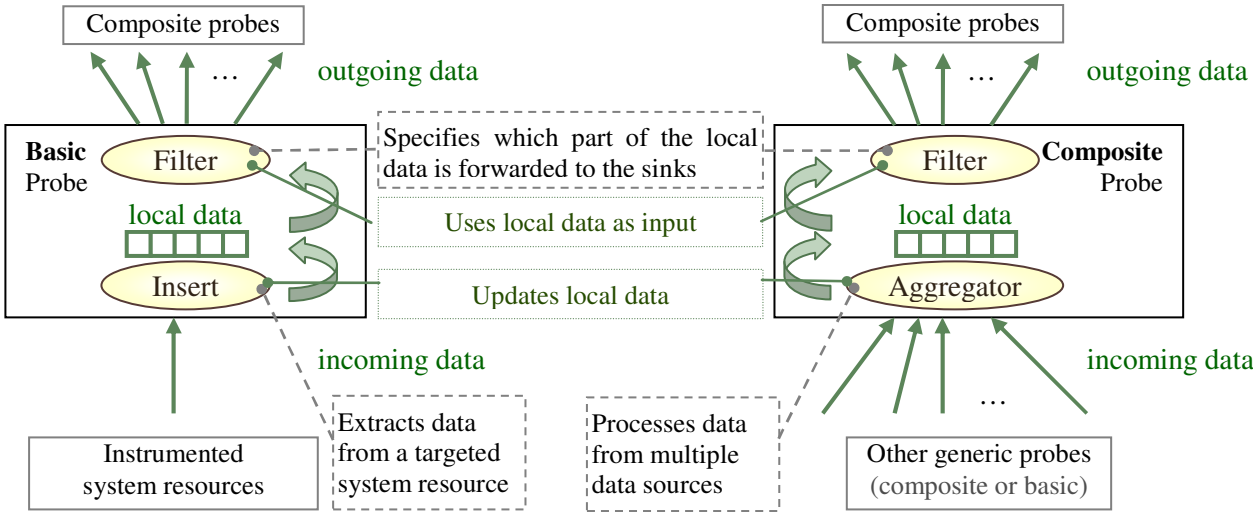

Figure 4: Data flow through Basic and Composite Probe
Basic Probes use an Insert component to extract monitoring data from system resources
Composite Probes use an Aggregator to process data from multiple sources
Basic and Composite Probes use a Filter to process local data into outgoing data for data sinks

2.4.2 Control flow

The control flow transports control commands targeted at managing the Probes' lifecycles. Such control commands can dictate a Probe's initialisation, starting, stopping, pausing, resuming or terminating. Control commands are targeted at a certain Probe and can be propagated to uniformly affect the targeted Probe's sub-graph. As such, control commands can be forwarded from Composite Probes at higher hierarchical levels down through the hierarchy to the Basic Probes level. This process is detailed as follows.

All control operations receive two common parameters, in addition to any operation-specific ones. These are the `target Id` and the `propagate` directive. The target Id uniquely identifies the targeted Probe to which the control command is addressed. This implies that a Probe receiving a control command is not necessarily the one that will execute the command. A control operation will be forwarded from a receiving Probe down through the monitoring hierarchy until reaching its targeted Probe, as identified by the target Id. Routing information stored in the Probe's Delegation Managers (subsection 3.5) is used to route a control command to its targeted Probe. The specific manner in which Probe Ids (or Blade Ids) and routing information is implemented in the current Component Probes version is described in section 2.2.

The second common control parameter is the propagation directive. This parameter indicates whether or not the control command should be propagated through the sub-graph that has the targeted Probe as root. If the propagate value is true, the control command will be executed on the targeted Probe, as well as on its child Probes, recursively.

2.5 Logical Architecture

The way Basic and Composite Probes are logically connected to form a monitoring hierarchy is depicted in Figure 5. Basic Probes are used to extract monitoring data from the managed system resources. Composite Probes are used to collect and

manage data from multiple Composite or Basic Probes. If a Probe C1 is sending data to another Probe C2, we consider that C1 is a *child* Probe with respect to C2 and C2 is a *parent* Probe with respect to C1. Basic Probes can only be children to other Composite Probes. Composite Probes can be parents to other basic or Composite Probes and can also be Children for other Composite Probes.
Monitoring data received at each Probe is locally processed and subsequently forwarded to the Probe's parents (i.e. *push* data transfer). Monitoring data is consequently propagated upwards throughout the Probe hierarchy, from the lowest Basic Probes level up to the root Composite Probes level. Additionally, clients can request a Probe to return its local monitoring data (i.e. *pull* data transfer).
Clients can send control commands targeted at a certain Probe. Control commands can be configured to be recursively propagated to the targeted Probe's children. In this case, the control command will uniformly affect the targeted Probe as well as the sub-hierarchy that has the targeted Probe as root.
As shown in Figure 5, clients can interact with Probes at any hierarchical level and not only via the root Probes. This means that a client can directly interact with any Probe in the hierarchy for requiring monitoring data or sending control commands.

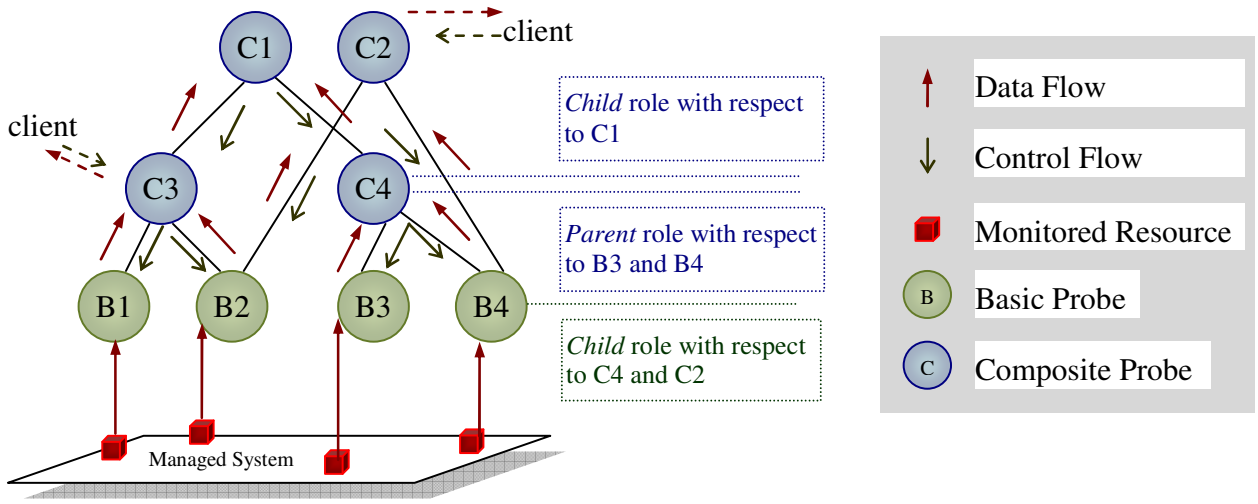

Figure 5: Logical architecture of a Composite Probes hierarchy

2.6 Probe interconnection and hierarchy types

Various solutions can be envisaged for implementing the logical Probe hierarchy discussed in the previous section (2.5 Logical Architecture). Solutions differ in the way Basic and Composite Probes are connected together to form a monitoring hierarchy.
The current Composite Probes implementation is based on the Fractal[1] component model. In this context, Probes are implemented as Fractal components and interconnected via Fractal inter-component bindings. This allows both local inter-Probe communication via direct method calls, as well as distributed communication based on Fractal RMI. Other communication protocols (e.g. based on JMX or JMS) can be used to support the inter-Probe communication. To implement this scenario, the Probe subcomponent in charge of managing the Probe's communication must be implemented to support the desired protocol (subsection 3.5 The Delegation Manager).

[1] The Fractal open-source project: fractal.objectweb.org

Different Probes in a hierarchy can be configured so as to use different communication protocols.

Considering the Fractal-based implementation approach, two possible solutions have been identified for designing the Probe hierarchy architecture (Figure 6). The first solution is based on *encapsulating* child Probes into the parent Probes (Figure 6-a). In case a child Probe has multiple parents all parent Probes will share the child Probe. This implies that a root Probe component will recursively contain all Probes in the monitoring hierarchy to which it is connected. Considering the example depicted in Figure 5, the Probe C1 will contain Probes C3 and C4, which will in turn contain Probes B1 and B2, and B3 and B4, respectively. Probe C2 will similarly contain probes C3 and C4, sharing them with Probe C1. The encapsulation approach takes advantage of the Lifecycle management mechanisms inherent to the Fractal component model. This means for example that starting or stopping all Probe components in a monitoring hierarchy would simply imply calling the `startFc()` or `stopFc()` method on the root Probe. The main disadvantage of this method is that client calls to Probes placed lower in the hierarchy should, at least initially, pass through all the parent Probes encapsulating the targeted Probe. In the example in Figure 5, the client accessing Probe C3, should initially pass via the root Probe C1. Another disadvantage of the encapsulation approach is a conceptual, architectural one. It is linked to the fact that a Probe would contain both the subcomponents that implement its functionality as well as its child subcomponents, all mixed together at the same level (e.g. Figure 8).

A second solution is to place all Probes at the same architectural level and link them together in a flat hierarchy (Figure 6-b), similar to the one depicted in the logical architecture in Figure 5. This approach has the advantage of allowing clients direct access to any Probe in the hierarchy. In the example in Figure 5, a client can directly access Probe C3, without having to pass through its parent Probe(s) first. Additionally, in this approach each Probe will only contain subcomponents needed to implement its functionality. This solves the architectural matter present in the encapsulation approach. The main disadvantage of this approach is that it cannot use the inherent Lifecycle management functionalities of the Fractal component model. Therefore, starting or stopping a monitoring hierarchy will imply individually calling the `startFc()` or `stopFc()` operation on every Probe in the hierarchy. Nonetheless, Composite Probes already provide a forwarding mechanism for propagating control commands, such as init, start or stop, through the monitoring hierarchy (sections 2.4.2 and Figure 18). Hence, the same mechanism can be similarly employed to propagate `startFc` and `stopFc` lifecycle commands, when the flat hierarchy approach is used.

The current Composite Probes implementation uses Fractal bindings for Probe interconnection. Both local and distributed communication scenarios are supported via direct method calls and Fractal RMI, respectively. The encapsulation approach has been initially selected in the current implementation.

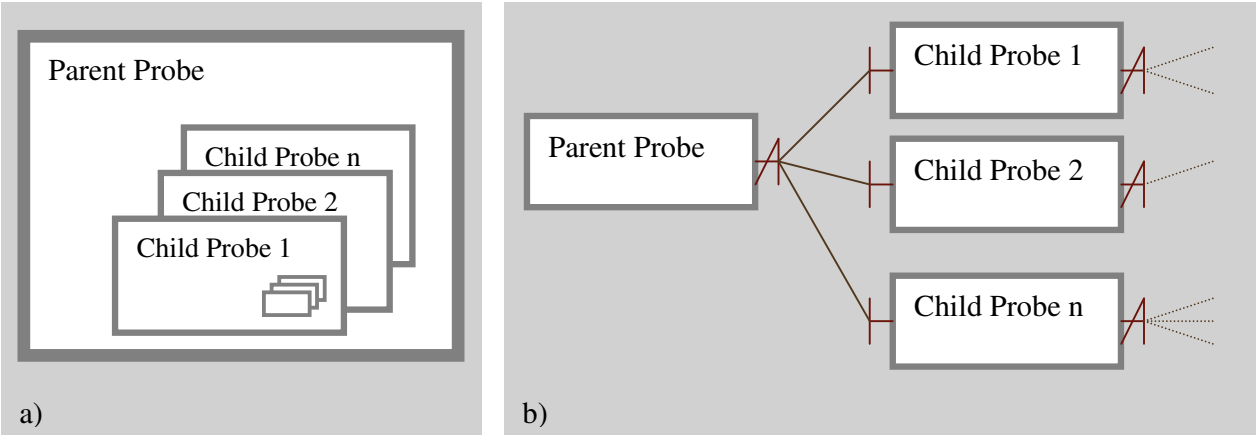

Figure 6: Fractal hierarchy types
a) encapsulation approach: parent probes recursively *contain* child probes
b) flat hierarchy approach: parent probes are *linked* to child probes

2.7 Composite Probes as a CLIF extension

In this section:

- ⇨ compare Composite Probes roles /architecture /functions with CLIF
- ⇨ describe the parts that have been reused and/or extended from CLIF
- ⇨ Discuss back-compatibility with the CLIF probes

3 Composite Probes Architecture

3.1 Architecture Overview

Basic and Composite Probes provide identical external interfaces and feature similar internal architectures.

3.1.1 Probe interfaces

The main external interfaces equally provided by Basic and Composite Probes are as follows:

- Server interfaces:
  - *Blade Management*: used by administrators to perform Probe management operations. Such operations include setting or configuring the Probe's aggregator, filter or scheduler functionalities (section 3.4)
  - *Data Collector Administration*: used by clients to requests monitoring data or meta-data on the managed resources
  - *Blade Control*: used by administrators to perform control operations such as init, start, pause, resume, or stop on a Probe
- Client interfaces
  - *Data collector write delegate*: used to forward data events to parent Probes
  - *Blade response delegate*: used to send asynchronous responses to parent Probes. These include notifications of abnormal situations such as the unexpected termination of a Probe operation.
  - *Supervisor information*: used to send Probe state information to parent Probes. This is typically information on the current Probe's state.

NOTE: additional client interfaces would be needed in case Probe hierarchies were built using the flat-hierarchy approach, instead of the current encapsulation approach (section 2.6):

- *Data collector administration delegate*: used to forward data collector administration requests to child Probes. This function is needed when a Probe receives an administration request that is not targeted at the current Probe.
- *Blade control delegate*: used to forward and propagate control commands to child Probes. This function is needed when a Probe receives a control command that is addressed to a different targeted Probe. It is also needed when a Probes receives a control command targeted at this Probe, with the propagate directive set to true. This means that the Probe must execute the command and forward it to all child Probes.

3.1.2 Probe subcomponents

The main architectural difference between the two probe types results from their different roles. Basic Probes are in charge of actually extracting monitoring data from the managed system. They use an *Insert* subcomponent for this purpose. Composite Probes are in charge of managing monitoring data received from multiple lower-level Probes. Consequently, Composite Probes will not use an Insert subcomponent themselves, but use *connections* to the lower-level Probes they must manage instead. In the current implementation approach, Composite Probe hierarchies are built by means of Fractal component encapsulation. This means that Composite Probes at a certain hierarchical level will *contain* the Probes at the immediately lower hierarchical

level. The possible ways of connecting Composite Probes and constructing monitoring hierarchies are discussed in section 2.5.
Apart from this difference, Basic and Composite Probes contain identical subcomponents. Evidently, the way some of these subcomponents inter-communicate is influenced by whether the Probe contains an Insert subcomponent or only connections to lower-level Probe components (Figure 7 and Figure 8).
The main Probe subcomponents common to both Basic and Composite Probes are as follows:

- *Data Collector* (section 3.4): is the first to receive incoming monitoring data; its role is to perform the initial data processing during runtime. Some of the main Data Collector functions include limited statistical calculations on the most recent data events; sending data events to the persistent storage support; and forwarding data events to subscribed clients (e.g. parent Probes).
- *Blade Adapter* (section 3.6): manages asynchronous communication for the Probe's control commands. Such commands dictate the Probe's initialisation, start, pause, resume or stop. More precisely, a client that sends a control command to a Probe will receive a response immediately, before the control command has been executed. Upon the control command's completion, the client will receive an asynchronous response, signalling the success or failure of the command's execution. This prevents a Probe's clients from being blocked while their control commands are being executed.
- *Storage Proxy* (section 3.7): manages monitoring data persistence, using a certain storage support (e.g. a relational database, or s file system). Incoming monitoring data is initially received and processed by the Data Collector. The Data Collector subsequently sends the incoming monitoring events to the Storage Proxy, which will send them to the persistent storage used. Persisted monitoring events can be used for further processing and complex statistic analysis. A different Storage Proxy implementation will be needed for each different persistent storage support used.
- *Blade Management*: represents a portal for all management operations on the Probe. Such operation can include setting a particular aggregation algorithm, a filtering criterion, or a certain time interval in a scheduling function. The Blade Management component will direct management operations to the specific component to which the operation is actually addressed (e.g. Data Collector or Blade Adapter component). This approach allows each Probe to provide a single management interface, thus shielding external clients from the Probe's internal architecture details.
- *Child and Parent Delegation Managers* (section 3.5): manage all Probe's communication with child and parent Probes, respectively.

Additionally, Basic Probes contain an extra subcomponent, specific to each monitored resource type:

- *Insert* (section 3.8): extracts the actual monitoring data from the managed system resources. Evidently, a different Insert implementation is needed for monitoring different resource types (e.g. different Insert implementations will be used to monitor a UNIX system's CPU and a Windows system's memory).

The presented Probe subcomponents are described in more details in sections 3.4 The Data Collector Component, 3.5 The The Delegation Manager Components, 3.6 The Blade Adapter Component, 3.7 The Storage Proxy Component and 3.8 The Insert

Component. The following two sections (3.2 and 3.3) present the general internal architectures of Basic and Composite Probes. They indicate the way the main Probe subcomponents are linked together to provide the Probe's functionality. The Storage Proxy component is not used in the current Composite Probes implementation and is therefore not depicted in the Probe architecture diagrams.

3.2 Basic Probe Architecture

Figure 7 shows the Basic Probe subcomponents, their main interfaces and the way they are bound to each other and to the Probe's external interfaces.

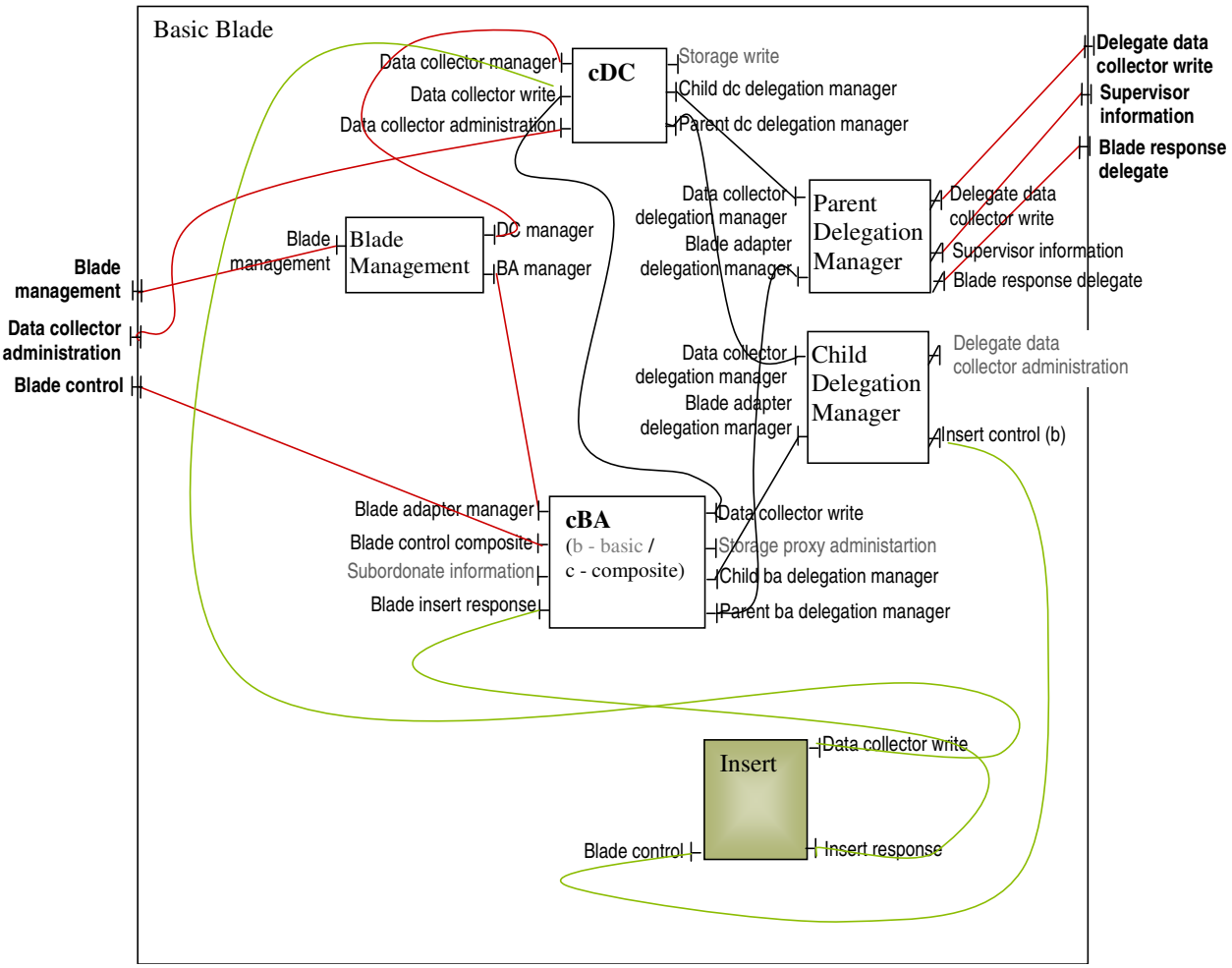

Figure 7: Basic Probe architecture

3.3 Composite Probe Architecture

Figure 8 shows the Composite Probe subcomponents, their main interfaces and the way they are bound to each other and to the Probe's external interfaces.

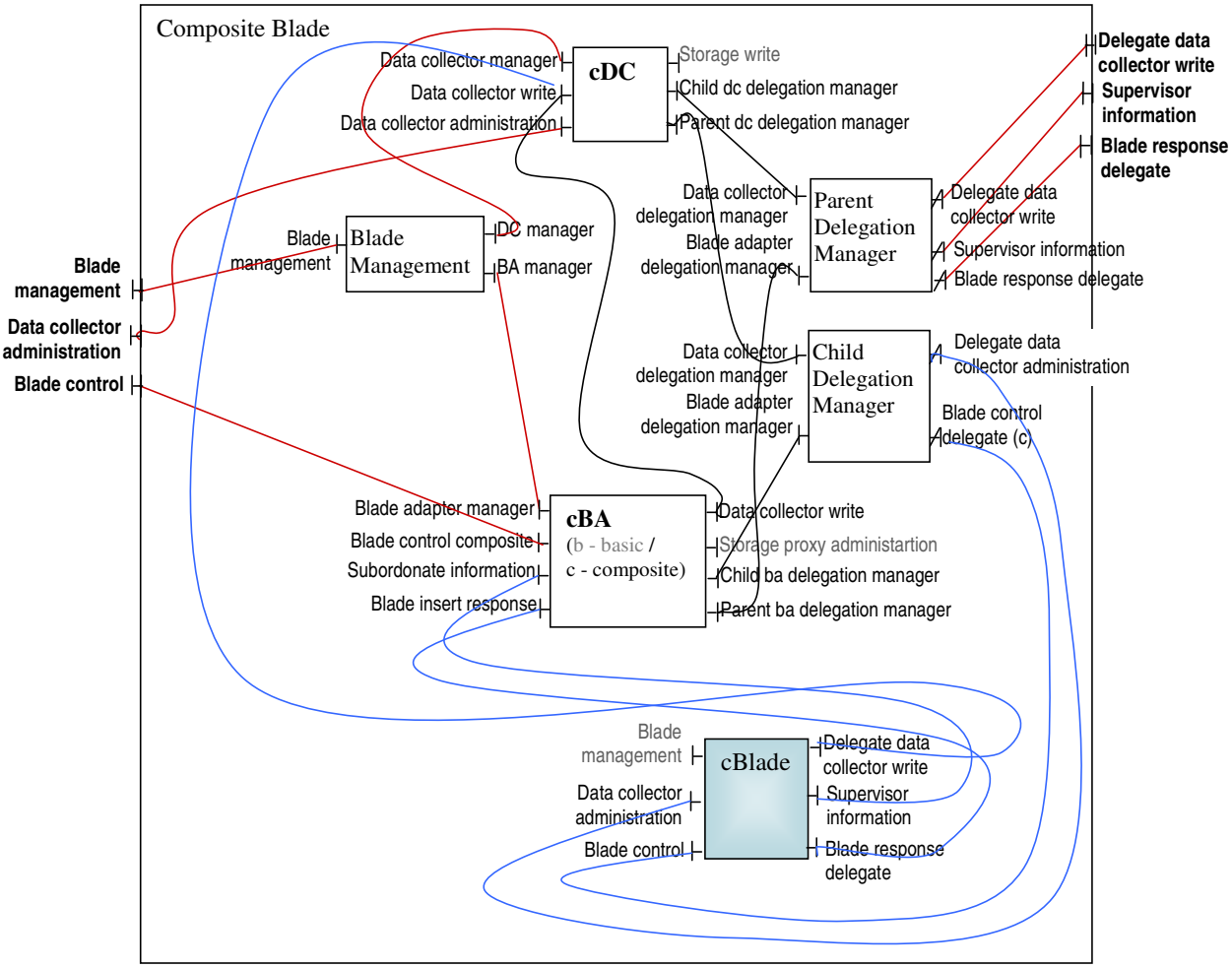

Figure 8: Composite Probe architecture

3.4 The Data Collector Component

Figure 9 shows the overall architecture of a Data Collector component. It depicts the main external and internal interfaces, subcomponents and bindings. The Figure also shows how a Data Collectors interacts with the child and parent Delegation Managers in order to communicate with child and parent Probes, respectively.

The main external interfaces provided by a Data Collector are as follows:

- Server interfaces:
  - *Data collector management*: used by administrators to configure the data collector, by setting and tuning its aggregation, filtering and scheduling functions
  - *Data collector write*: used by data sources to send new monitoring events. Data sources consist of Interceptors in Basic Probes and of child Probes in Composite Probes.
  - *Data collector administration*: used by Probe clients to request monitoring data and meta-data on the managed resources.
- Client interfaces:
  - *Parent delegation manager*: used for all communication with parent Probes
  - *Child delegation manager*: used for all communication with child Probes

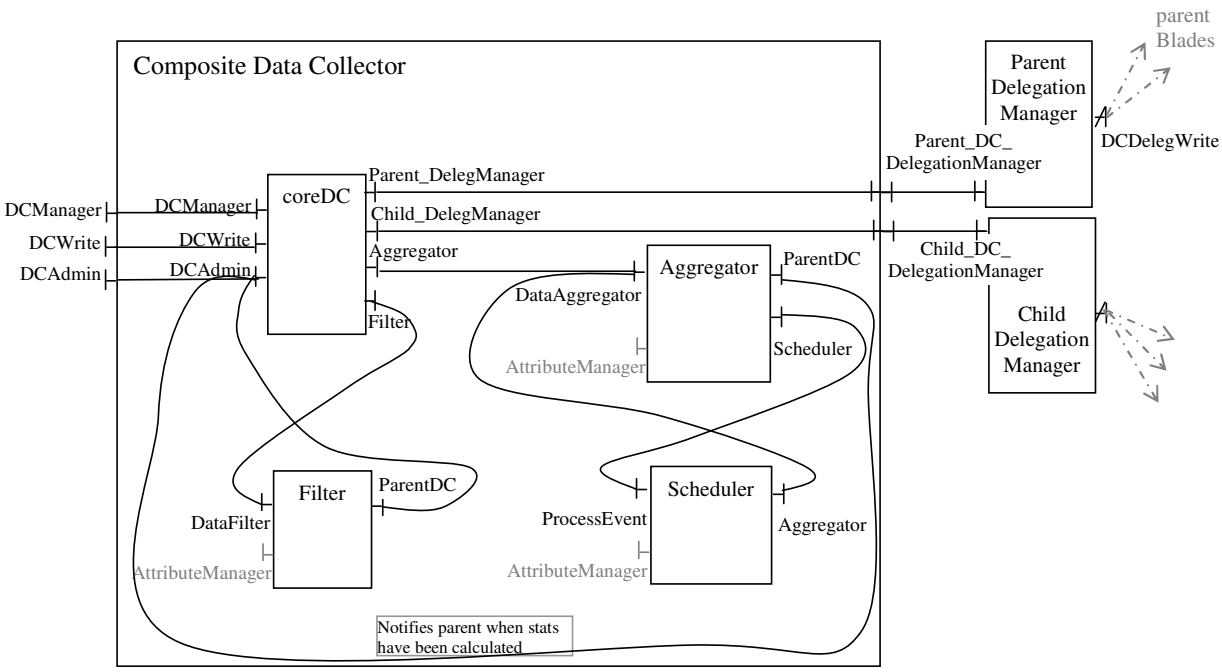

Figure 9: Data Collector internal architecture and delegation management

3.4.1 Aggregator

The *Aggregator* is a configurable data-processing subcomponent of the `DataCollector` component of each Composite Probe (Figure 9). Multiple Aggregator types can be available and each Composite Probe can be configured with a different aggregator type.

Role

The Aggregator's role is to manage incoming data from multiple data sources, or child probes (Figure 3 and Figure 4). It specifies how data from multiple sources is being processed in order to calculate the probe's local data (Figure 10).

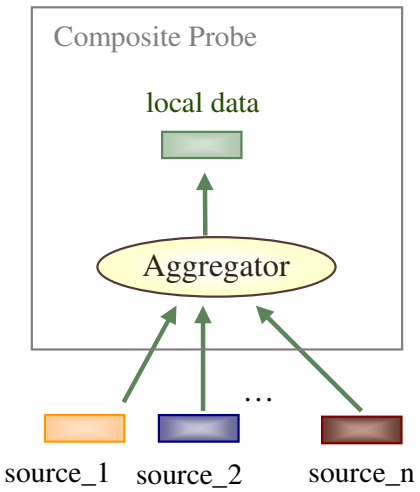

Figure 10: General data flow in a composite probe Aggregator:

The Aggregator processes incoming data from multiple sources and stores the result as local data

Aggregator Types

An aggregator's data management policy will depend on two major factors.
The first factor is the desired processing function to be applied on the incoming data. This function is used for calculating summary statistics, which are subsequently stored as local data. Possible data processing functions can include various mean or median functions, standard deviation, variance, minimum or maximum functions.
The second factor is related to the aggregator's data sources types. In one possible scenario, all aggregator's data sources are of the same type. This means that all data sources send the same type of data, with similar formats and identical semantics. In this case, the aggregator can merge the data received from the different sources together and apply its processing function over the entire data set. An example of this scenario can be a cluster CPU aggregator that receives data from multiple CPU probes. In this example the aggregator calculates the overall cluster CPU load, based on data received from CPU probes installed on every cluster machine.
In a second scenario, an aggregator's data sources are of different types. This means that various data sources send different types of data, in different formats and/or with different semantics. In this scenario, the aggregator must *separately* process each data set that it receives from each different data source. An example of this scenario can be an aggregator that calculates a system's general load based on data received from various system resource probes. In this scenario, a different probe type will be used to monitor each system resource type. This implies that the load aggregator will receive data from different probe types, such as memory probes, CPU probes, disk probes and network probes. In this scenario, the aggregator should separately treat data received from its different sources, rather than merging all data together.
Two aggregator types are provided in the current Composite Probes implementation (Figure 11):

- `ManyPerIdenticTypeAggregator`: used to aggregate data from identical data sources. Each measure will be a long[] containing multiple parameter values. For example, a CPU probe will send values for the %CPU, %CPU user and %CPU kernel parameters, with each new measurement. All data sources must send the same parameters and in the same order (
- Figure 1).

- `OnePerTypeAggregator`: used to aggregate data from different source types. Each measure can contain a single parameter value, formatted as a long[] type of size one (
- Figure `1`).

Aggregator Interface

The most *important methods* provided by an aggregator class include (Figure 11):

- `addMeasure( IdentifiableProbeEvent newMeasure )`: called by a data source to send a new measure. The IdentifiableProbeEvent parameter includes the source's identification information (i.e. unique id) and the new data values.
- `calculateStats()`: calculates the summary statistics and stores the result as local data. Statistics are calculated accordingly to the aggregator's specific function and are based on data collected in the aggregator over the most recent interval (3.4.3 Scheduler).
- `getCalculatedStats()`: called by the aggregator's clients in order to obtain the local monitoring data. The local data returned represents the summary statistics calculated over the most recently completed interval. All such requests made over a certain interval will receive the same local data values, as calculated at the end of the previous interval.

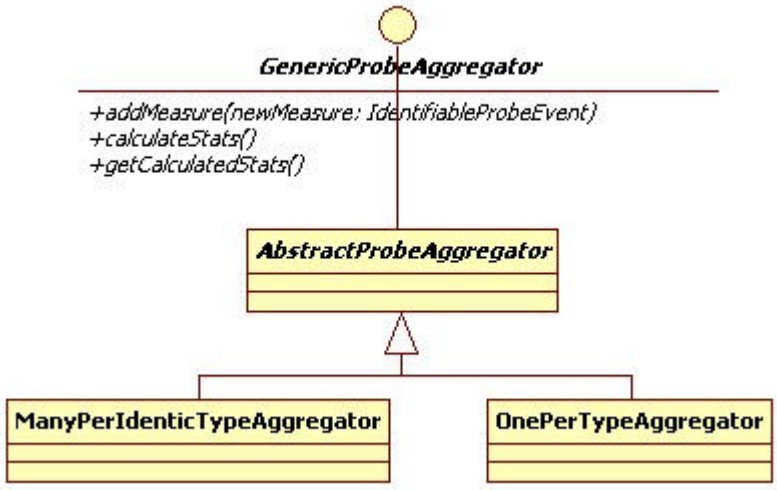

Figure 11: Aggregator class diagram

Extensibility

Additional aggregator types can be implemented and used for data processing in Composite Probes. The sole restriction is that all aggregators must implement the `GenericProbeAggregator` interface (Figure 11).

3.4.2 Forwarding Filter

The *Forwarding Filter* (or Filter) is a configurable data-processing subcomponent of the `DataCollector` component of each Composite Probe (Figure 8 and Figure 9). Multiple Filter types can be available and each Composite Probe can be configured with a different filter type.

Role

The Forwarding Filter's role is to determine the outgoing data to be sent to the probe's data sinks, or parent probes (Figure 3 and Figure 4). It specifies how the local data is being processed for preparing the probe's outgoing data (Figure 12).

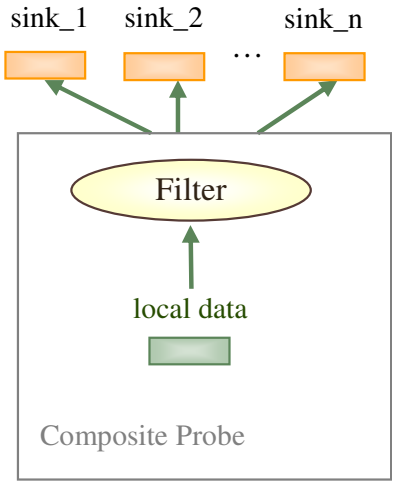

Figure 12: General data flow in a composite probe Filter:

The Filter processes the local data to obtain the outgoing data for the probe's sinks

Forwarding Filter Types

A typical filter determines which subset of the input data will be sent as output data. Different filter types can be implemented to specify different filtering policies. In the Composite Probes context, a filter will always use the local probe data as input and provide a filtered data as output. The filter is unaware of whether the output data will be sent out to one or multiple data sinks. All inter-probe communication is managed by the probe's Delegation Manager instead (subsection 3.5 The Delegation Manager). Two filter types are available in the current Composite Probes implementation, as shown in Figure 13:

- `SingleParamFilter`: this filter selects a single data element from the input data set and sends this element as the output data set. The index used to select the exact data set parameter is a configurable attribute.
- `TransparentFilter`: this filter has no effect, meaning that the entire input data set is sent unchanged as the output data set. This filter has been implemented in order to maintain an uniform data processing path through all Composite Probes, irrespective of their specific, internal configuration.

Forwarding Filter Interface

The most important methods implemented by a forwarding filter include:

- `filterStats( IdentifiableProbeEvent probe )`: filters the local probe data received as parameter conforming to its filtering policy
- `getForwardedLabels()`: returns the filtered subset of labels corresponding to the filtered subset of data that this filter provides as output. Parent probes that receive this probe's filtered data will use this method to determine the

exact labels (monitored parameter names) corresponding to that data (monitored parameter values).

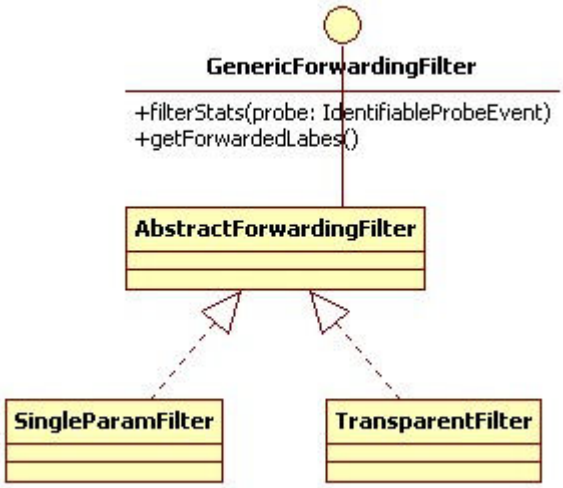

Figure 13: Forwarding Filter class diagram

Extensibility

Additional filter types can be implemented and used for data processing in Composite Probes. The sole restriction is that all filters must implement the `GenericForwardingFilter` interface (Figure 13).

3.4.3 Scheduler

The *Scheduler* is a configurable subcomponent of the `DataCollector` component of each Composite Probe (Figure 8 and Figure 9). The Scheduler's exact functioning can differ between different Scheduler implementations. Each Composite Probe can be configured with a different Scheduler type.

Role

The Scheduler's role is to dictate the Composite Probe's data processing *intervals*. Incoming data events are being collected over each interval. At the end of each interval, all collected data is used to calculate the Composite Probe's summary statistics. The Aggregator (3.4.1) is in charge of performing the statistical calculations on data events collected over the most recently completed interval. The collected data events used to calculate the local statistics are subsequently deleted. The calculated statistics are sent out to the probe's data sinks, or parent probes (Figure 3 and Figure 4). In summary, the Scheduler dictates the intervals at which collected data is used to calculate local statistics, which are subsequently filtered and forwarded to parent probes.

Each Scheduler has an associated *Task*. The Scheduler will trigger the Task's execution at the end of each interval.

Scheduler Types

The exact manner in which a Scheduler determines the Composite Probe's intervals depends on the scheduling policy it implements. All schedulers are aware of new data events being received. Each specific Scheduler implementation can choose whether or not to use this information. Likewise, the Scheduler's Task can be implemented to trigger various data processing or forwarding functions.

Two Scheduler types are available in the current Composite Probes implementation (Figure 14):

- `ThresholdScheduler`: determines intervals based on the number of monitoring data events received by the probe. This scheduler uses a configurable threshold to determine the end of each interval. For example, a threshold set to a value of three means that the end of each interval is signaled each time three data events have been received. The number of received events is reset at the beginning of each interval.
- `PeriodicScheduler`: determines intervals on a strict time basis. This scheduler implementation ignores all information on new incoming data events. The Scheduler uses a configurable time period to determine the end of each interval.

A single Task is available in the current Composite Probes implementation:

- `StatsCalculatorTask`: triggers the Aggregator's statistical calculation function. This means that at the end of each interval all data events collected during the completed interval are used to calculate the local probe data. In the current implementation, calculated statistics are immediately filtered and forwarded to the probe's parents.

Current Implementation

The most important methods provided by a Scheduler include:

- `setTask()`: sets the task that the scheduler will trigger at the end of each interval
- `propertyChange( newEvent )`: signals the Scheduler the arrival of a new monitoring data event. The new event parameter contains the received monitoring data values. The way this event is used depends on the specific implementation of each Scheduler type.

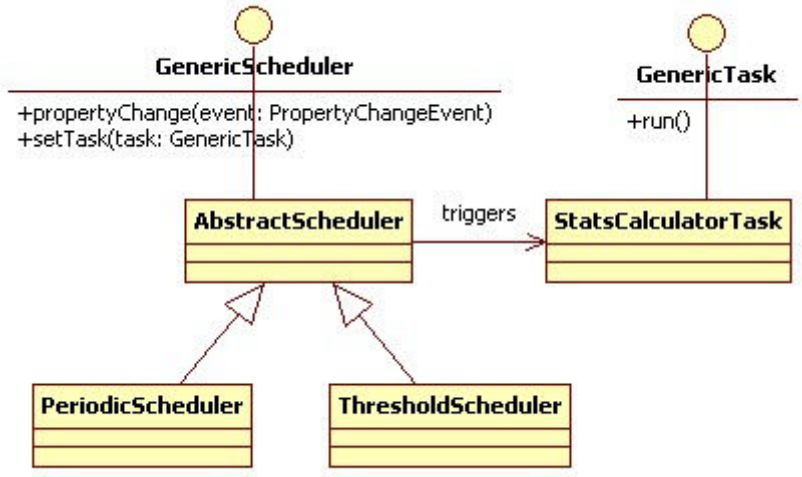

Figure 14: Scheduler class diagram

3.5 The Delegation Manager Component

Role

The role of a Delegation Manager of a certain Composite Probe is to handle any communication with entities external to that Composite Probe. In the current implementation Delegation Managers communicate exclusively with other Composite Probes, which are whether children or parents to the Delegation Manager's own Composite Probe.

Delegation Manager Types

Two main factors determine the different types of Delegation Managers.
The first factor is the communication support the Delegation Manager uses to perform the actual communication with the external entities. The current Composite Probes implementation uses Fractal component bindings all inter-Probe communication. Consequently, the communication support is based on method calls for local communication and on Fractal RMI for distributed scenarios. Additional Delegation Managers can be implemented to provide support for further communication protocols such as JMS- or JMX-based.
The second factor is the type of communication the Delegation Manager must mediate. This factor is defined by the type of messages the Delegation Manager must be able to handle (i.e. receive and send). Based on this criterion, Delegation Managers can be classified as follows (Delegation Manager interfaces in Figure 15):

- Data Collector Delegation Manager for parent communication (`ParentDCDelelegationManager` interface): handles communication initiated by the Data Collector and targeted to the Composite Probe's parents. This includes for example the transmission of filtered data to the Composite Probe's parents.
- Data Collector Delegation Manager for child communication (ChildDCDelelegationManager interface): handles communication initiated by the Data Collector and targeted to the Composite Probe's children. This includes the transmission of client data requests that are targeted at lower-level Composite Probe in the hierarchy. In the example in Figure 1, a client can ask the Cluster Probe to return the data of the CPU1 Probe. The client will specify the unique ID of the CPU1 Probe in their request. In this scenario, the Cluster Probe will determine that it is not the targeted Probe for this request and will use its corresponding Delegation Manager to forward the request to the successor Probe en-route to the destination, in this case the System1 Probe. The System1 Probe will follow the same process to forward the request to the targeted CPU1 Probe.
- Blade Adapter Delegation Manager for parent communication: handles communication initiated by the Blade Adapter and targeted to the Composite Probe's parents (`ParentBADelelegationManager` interface). This includes for example control commands such as init, start or stop probe.
- Blade Adapter Delegation Manager for child communication (`ChildBADelelegationManager` interface): handles communication initiated by the Blade Adapter and targeted to the Composite Probe's children. This includes asynchronous response messages such as system information alarms.

Two Delegation Managers were defined in the current implementation to meet the aforementioned communication requirements. These are the `ChildDelegationManagerImpl` and `ParentDelegationManagerImpl` classes, which are responsible for all communication with the child and parent Composite Probes, respectively. Based on this present design, Delegation Managers of different types can be implemented and used so as to allow a certain protocol be used for communicating with parent probes and a different protocol for communication with children probes (Figure 15).

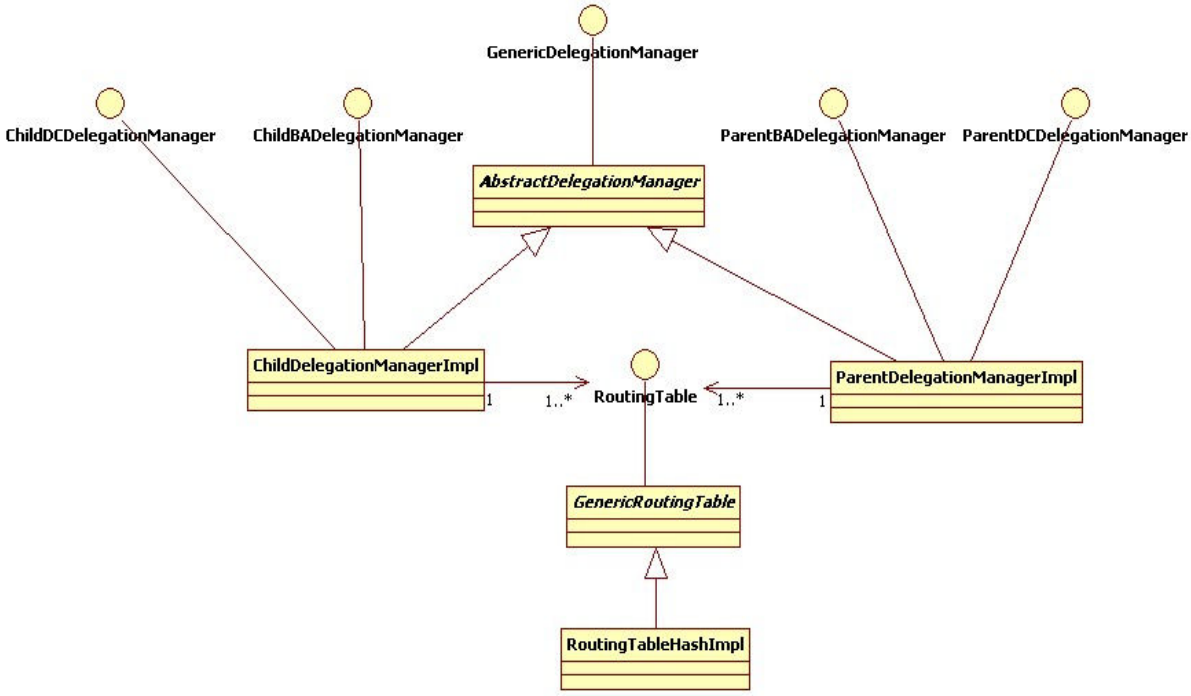

Figure 15: Delegation Manager class diagram

Routing Table

Delegation Managers use Routing Tables for managing information on the targeted external entities with which they communicate. This information includes the identities and locations of neighbouring external entities. In case a targeted node is not directly accessible from the current node, a Routing Table determines the successor node on the path en-route to the targeted node. In the example scenario in Fig X Routing Tables at each node are used to root a request received at the Cluster probe, requiring information from the CPU1 probe. In this example, the request will be routed from the Cluster probe to the System1 probe and finally to its actual target, the CPU1 probe.

The current Composite Probes implementation uses Fractal component bindings for inter-Composite Probes communication. Consequently, the Routing Table implementation manages the associations between the ids and interface references of the neighbouring Composite Probes. A Hashtable object is used as storage support for these associations.

3.6 The Blade Adapter Component

Figure 16 depicts the Blade Adapter's interfaces and the way they are connected to the child and parent Delegation Managers, for communication with child and parent Components, respectively. The general roles of these interfaces are as follows:

- Server interfaces
  - *Blade adapter management*: used by administrators to configure the Blade Adapter
  - *Blade control*: used by administrators to send control commands to the Blade Adapter. Such commands include the initialisation, start or stop of the Probe.
  - *Subordinates information*: used by child Probes to send state information or alarms
  - *Insert response*: used by child Probes to send asynchronous responses, typically signalling an abnormal or unexpected event
- Client interfaces:
  - *Parent delegation manager*: for all communication with parent Probes
  - *Child delegation manager*: for all communication with child Probes

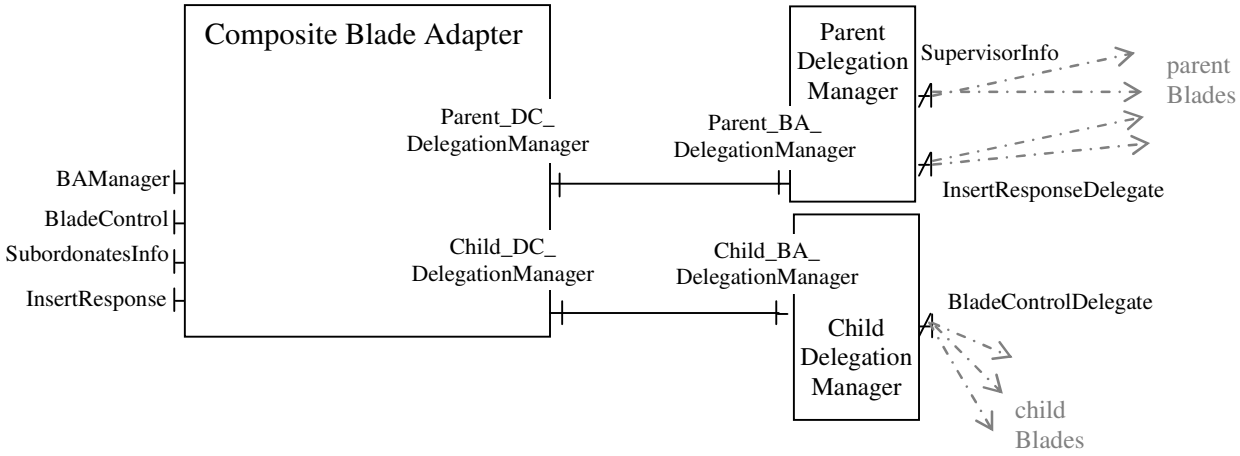

Figure 16: Blade Adapter architecture and delegation management

3.6.1 Threading considerations

In this subsection:

⇨ One Thread per Blade Adapter instance - one Blade Adapter instance per Composite Probe

⇨ Note: Another thread per Scheduler – Scheduler instances can be shared among several Composite Probes

3.7 The Storage Proxy Component

This section will discuss:

⇨ Main Storage Proxy interfaces and subcomponents

⇨ Currently not implemented in Composite Probes

⇨ Will reuse Storage Proxy implementation of CLIF

3.8 The Insert Component

Insert components must have the same type (i.e. provided and required interfaces) irrespective of their provider and the type of system resource they monitor. Clearly, the Insert component's implementation will vary with different providers and monitored system resource types.

The main external interfaces of the Insert component are as follows:

- Server interfaces:
  - *Blade control*: used by the Blade Adapter to asynchronously forward control commands to the Insert component.
- Client interfaces:
  - *Data collector write*: used to send collected monitoring data to the Basic Probe's Data Collector (Figure 7)
  - *Insert response*: used to signal abnormal situations to the Basic Probe's Blade Adapter (Figure 7)

4 Information Flow Implementation

4.1 Data Flow Implementation

Figure 17 shows how new data events from an external data source is sequentially handled by the Data Collector's subcomponents. The data is subsequently forwarded to parent Probes, via the parent Delegation Manager.

New data events received by the Data Collector (DC) (i.e. the coreDC subcomponent of the DC) are immediately forwarded to the DC's aggregator. The aggregator first signals the new event to the DC's scheduler. The exact manner in which the scheduler will use this information depends on the particular policy the scheduler implements. The aggregator subsequently merges the new data event with the existing events already collected during the current interval. The exact manner the aggregation process is performed depends on the particular aggregation policy implemented. At the end of each interval, the scheduler executes its associated task, consequently calling the aggregator's calculation function. The aggregator calculates the Probe's summary statistics, based on all data events collected during the completed interval. Next, the aggregator signals the availability of new summary statistics to the core data collector. These statistics are then immediately filtered and forwarded to the parent delegation manager, which propagates them to subscribed parent Probes.

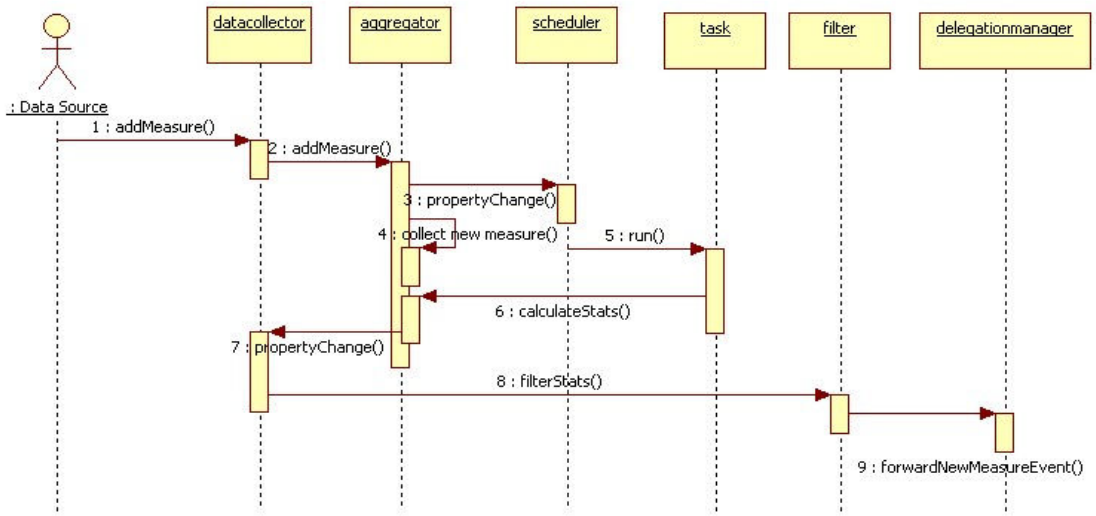

Figure 17: Data flow inside a Composite Probe - the Data Collector handles incoming monitoring evens, schedules intervals, calculates local statistics, filters outgoing data and forwards data to parent Probes

4.2 Control Flow Implementation

The way an init control operation is handled by a Composite Probe and propagated to its child probes is depicted in Figure 18. Composite Probes use a generic control mechanism for handling control commands. This means that the same methods are being called for handling all control commands, forwarding the specific command name and parameters as arguments. In the example in Figure 18, all generic control methods, such as `performCtrlOperation`, `do_operation`, or `propagate_control_operation`, receive an `OP_INIT` parameter that indicates that this is an init operation. As shown in the diagram in Figure 18, a Probe's Blade

Adapter initially sends an 'operation in progress' response to the client. The client is consequently unblocked and can perform other tasks while waiting for the control operation to complete. The Blade Adapter subsequently executes the control operation and sends an asynchronous response to the client upon completion.

All control operations receive two common parameters in addition to any operation-specific ones (section 2.4.2). These are the target Probe Id and the propagate directive. When executing a control command the Blade Adapter initially checks if it is the targeted Probe, by comparing its Bade Id (or Probe Id) with the targeted Id received. If this Probe is not the targeted Probe, then the Blade Adapter simply forwards the control command to the next Probe en-route to the targeted Probe. This is done via the Probe's Delegation Manager (subsection 3.5), which holds routing information and can communicate with neighbouring Probes. If the current Probe is indeed the targeted Probe, then the Blade Adapter verifies the propagate directive. If this parameter is true, the control command is first forwarded for execution to all the Probe's children. The current Probe subsequently waits until it receives all asynchronous responses indicating the control operation's completion on all the child Probes. The Probe evaluates the responses and if it 'considers them satisfactory' it proceeds to executing the control command itself. A specific function is used to determine the successful command execution on a Probe's sub-graph, based on the received completion responses. The current implementation considers an execution successful only if a successful completion response is received from all child Probes.

The sequence diagram in Figure 18 shows how an init control command is being handled by a Probe's Blade Adapter. The example scenario considers that the receiving Probe is indeed the targeted Probe and that the propagation directive is true. The diagram shows how the generic control flow management mechanism is used to implement the control command processing procedure described in the previous paragraph.

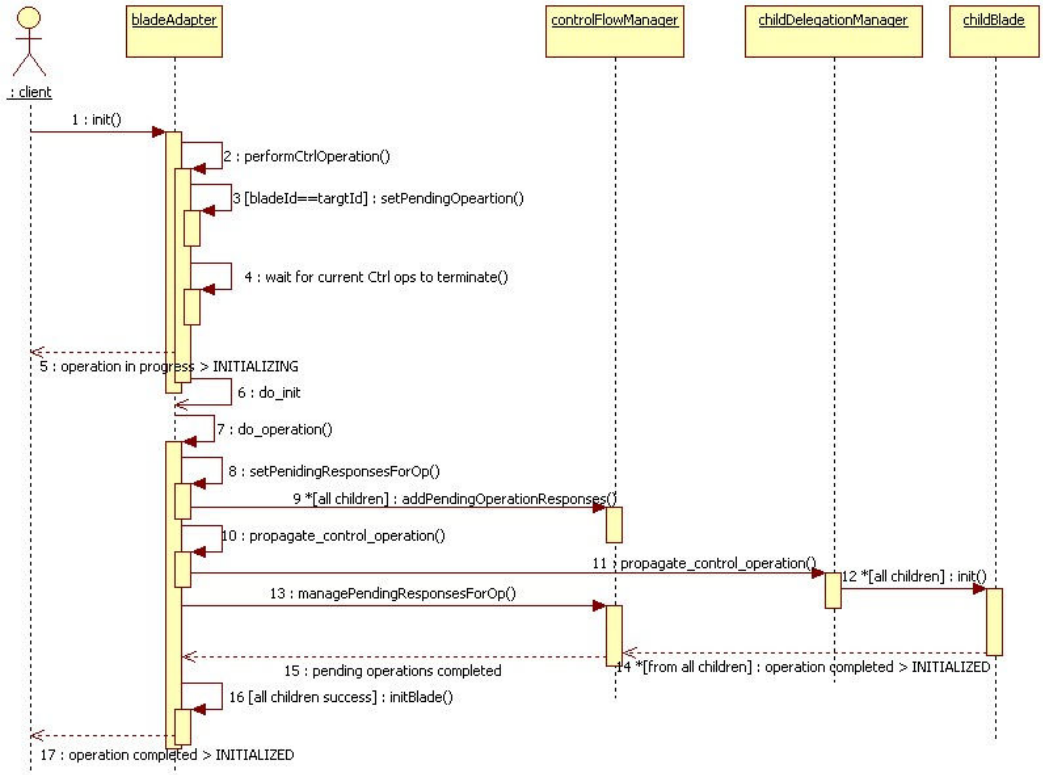

Figure 18: Control flow inside a Composite Probe – the Blade Adapter asynchronously handles incoming control operations using a generic mechanism: propagate control operation to children, wait for all child operations to complete, perform control operation locally and send operation-completed message to parent Probes

5 Composite Probes extensibility

The modular Composite Probes design allows the current implementation to be extended in various ways. The main extension points include the monitored resource types, the aggregation, filtering and scheduling functions, communication protocols and provided storage support.

5.1 Monitored resource types

The current Composite Probes implementation provides monitoring support for resources such as CPU, memory, network, disk and JVM, on Windows and UNIX systems. Additional Basic Probe types can be implemented and seamlessly integrated into the Composite Probes framework. For example, new probes can be added for monitoring application workloads, instance cache sizes, or thread and connection pools. The only requirement for new Basic Probes is that they must implement the external Probe's interfaces (subsection 3.1.1 Probe interfaces). Existing monitoring probes can also be integrated as Composite Probes, provided they are wrapped to provide the required interfaces. In case an existing monitoring probe cannot be modified, it can still be used as a semi-compliant Composite Probe. This means that the probe's provided monitoring data will be used to send data events to Composite Probes, while control commands will not be supported. In other words, semi-compliant probes can be used to provide monitoring data but cannot be controlled or configured.

5.2 Data processing procedures and functions

The current Composite Probes implementation provides basic aggregator, filtering and scheduling functions. Additional aggregator, filter and scheduler components can be implemented so as to extend the implemented data processing policies. The only restriction is that the new components must implement the corresponding aggregator, filter or scheduler interfaces (subsections 3.4.1, 3.4.2 and 3.4.3).

5.3 Inter-Probe communication protocols

The current Composite Probes implementation uses Fractal component bindings for inter-Probe communication. Composite Probes can be extended to support additional communication protocols, by implementing new Delegation Managers for those protocols (section 3.5).

5.4 Persistent storage support

The initial Composite Probe implementation will reuse the storage support and functions provided by the CLIF framework. This will be achieved by reusing the Storage Proxy component in CLIF, which provides persistent storage support based on a file system. Other storage support types, such as relational databases, can be added by implementing corresponding Storage Proxy components that can manage the new persistence types.

6 Future work

Several extensions can be envisaged to the Composite Probes framework, so as to support additional data formats and data-processing functions.

6.1 Additional data types and formats

The current Composite Probes implementation is limited to the `long[]` and `String[]` data types, for managing the values and names of monitored parameters, respectively (section 2.3 Data types). This approach can be extended so as to support additional information and/or additional data formats. As such, Composite Probes can be further developed to provide more comprehensive meta-data on the monitored resources. For example, a CPU monitoring Probe could provide information on the processor's type, producer and speed. This information would be provided in addition to the current information on the monitored parameter names and values, such as %CPU, %CPU user and %CPU kernel, which are generic to any CPU resource.
The additional meta-data proposed (e.g. Objects o xml format) would be indeed more expensive to transport to monitoring clients than the simple long[] values currently are. However, this meta-data will rarely change and will therefore rarely be updated. In the meantime, in some cases such meta-data can be crucial for a comprehensive analysis of the received monitoring values.

6.2 Flexible data-processing architecture

The Composite Probes architecture (sections 3.2, 3.3 and 3.4) can be extended so as to support more flexible data-processing constructs. In the current design, the data-processing path and architecture are fixed. More precisely, incoming monitoring data is received by the Probe's Aggregator and signalled to the Probe's Scheduler. The Scheduler will dictate when the Aggregator is to calculate summary statistics based on previously collected data events. Calculated statistics are subsequently filtered and forwarded to subscribed client and/or parent Probes. This procedure's flexibility is currently limited to specifying the precise aggregation, scheduling and filtering algorithms. However, architectural changes are not explicitly supported to allow more flexible data processing paths. Increased flexibility would allow specifying such data processing paths so as to involve several combined aggregators, sequentially linked with various filters and schedulers.

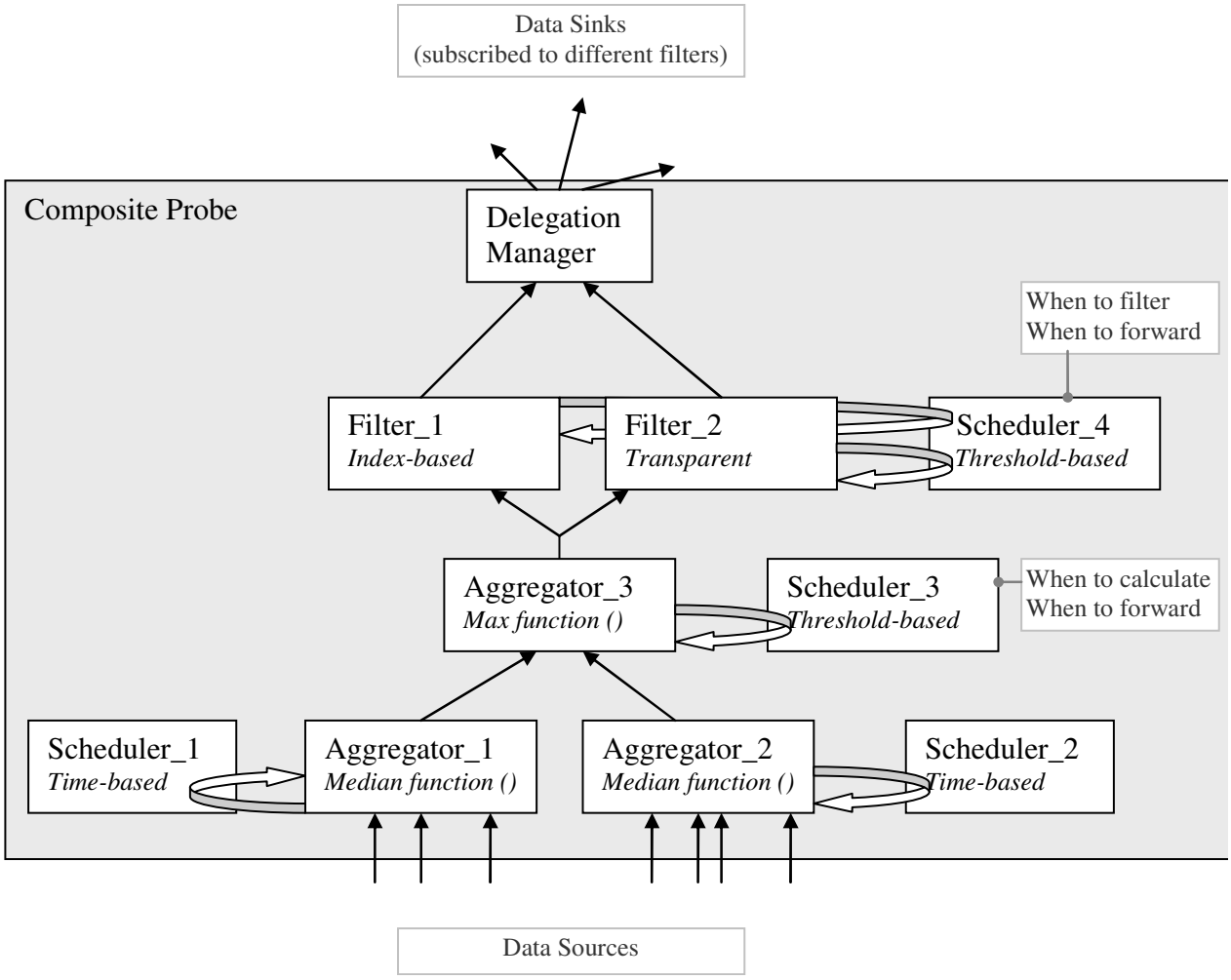

Figure 19: An arbitrary data-processing example based on a flexible Composite Probe architecture

6.2.1 Composite aggregators and filters

Two solutions are presently available for specifying such data-processing paths in the current Composite Probes implementation.

The first solution is to create one Composite Probe instance for every needed aggregator and obtain the desired combined aggregation function by correspondingly linking these Composite Probe instances together. The main disadvantage of this solution is that it may not scale well if complicated aggregation functions are needed for combining data from few child Probes. This is because full Composite Probe instances would be created to perform relatively simple aggregation procedures, which may introduce unnecessary overheads. Therefore, this solution's viability depends on the scale at which it is implemented and the overall impact of the introduced overheads on the monitoring system's performance.

The second solution is to implement custom aggregators that comply with the Composite Probes specification, so as to implement the desired composite aggregation function. Such aggregators can indeed be built by composing other aggregators, but this would be an ad-hoc design specific to each solution. The Composite Probes architecture can be extended to support building such aggregators

in a uniform manner, by simply specifying the desired combination of existing aggregator components.

6.2.2 Different filters for different subscribers

In this subsection:

- ⇨ Currently all data subscribers receive the same data.
- ⇨ Extend to allow local data to be processed by different filters and allow different subscriber groups receive data from different filters

6.2.3 Multiple schedulers

In this subsection:

- ⇨ Different schedulers for timing different data-processing and forwarding functions
- ⇨ A scheduler can dictate when data is to be calculated, when data is to be filtered and when data is to be forwarded
- ⇨ If flexible data-processing chains can be built out of linkable data-processing components, a separate scheduler can be attached to each data-processing component in order to determine when the component is to calculate and forward its processed data to the next component in the chain.
- ⇨ Scheduler instances can be shared

6.2.4 Flexible data-processing chains

In this subsection:

- ⇨ Data inputs,
- ⇨ Flexible linked-chain of aggregators, schedulers and filters
- ⇨ Data outputs