Reflective Quality of Service in the *Real-Time Performers* framework [⋆]

Andrea Trentini, Daniela Micucci

D.I.S.Co. - Università degli Studi di Milano-Bicocca
Via Bicocca degli Arcimboldi 8, 20126 Milano, Italy
`(trentini|micucci)@disco.unimib.it`

Abstract. The Real-Time Performers (RTP) is a Java framework to design distributed soft real-time systems based on timed plans. Timed plans - they define system workflow - contain actions to be executed by distributed components at specified times. The execution of a plan, however, has its costs, in terms of memory, power, bandwidth, etc. This paper presents RTP extensions to keep track (and take control) of these factors impacting on system Quality-of-Service. With these extensions the system itself is able to control its own behaviour by analysing its past history and by tuning the future part of its timed plan.

Keywords: quality of service, resource allocation, reflection, soft real-time.

1 Introduction

A time-sensitive system needs an underlying software architecture capable of capturing the temporal aspects belonging to the system itself. This kind of system needs to execute different activities with different, dynamic, and inter-dependent temporal requirements. Moreover it may happens that such kind of systems need to dynamically change the activities temporal requirements. To the knowledge of the authors, except for very recent efforts [7], current approaches to design of time-sensitive systems are based on outgrowths of classic approaches to the design and implementation of concurrent and real-time systems, and are not adequate to deal with the whole class of temporal aspects of computations at architectural level.

Real-Time Performers (RTP) is an architectural framework based on reflection that allows monitoring and control of the time related behavioural aspects of the systems built upon it. RTP is based on the following concepts:

- the system runs *temporally planned actions*;
- actions are planned for execution by placing them in *timelines* (defining the overall behaviour of the system);
- a timeline is "ticked" by a *virtual clock*;

[⋆] This work has been developed within the italian MURST-FIRB Project MAIS (http://black.elet.polimi.it/mais/)

– a virtual clock may be speed-tuned to modify the execution rates of actions on its timeline;
– a *strategist* may control the system by tuning virtual clocks an by changing the content of timelines (i.e. adding/removing/modifying actions).

To make RTP more adaptive (for limited environments such as small/mobile devices) and more self-aware, some extensions were required. There are applications for which the simple RTP time awareness is not enough and some degree of control over resource consumption is badly needed. For example, think about battery powered and limited hardware (memory and computing power) field monitoring devices such as networked/distributed surveillance cameras. These devices can be remote controlled, but they have far from infinite hardware resources and they take some time to process data, this is an example field for which RTP had to be adapted. New features to monitor and control "non-functional" parameters had to be added. *Non Functional Aspects* management is also known as Quality-of-Service (henceforth QoS) management. In RTP, QoS means self monitoring and (if possible) control resource allocation and performance. In other words, RTP QoS provides reflective mechanisms to let the system reason about its own performance [2] and (optionally) apply corrections.

The structure of this paper is the following: Section 2 describes RTP concepts, Section 3 describes extensions for time related QoS, and Section 4 describes resource QoS.

2 RTP concrete architecture

An application program to explicitly deal with non-functional requirements needs abstractions modeling these requirements so that they can be directly observed and controlled [6]. In this view, a time-sensitive system must explicitly deal with all the aspects concerning its temporal behaviour. RTP tries to capture and to model temporal abstractions into a reference software architecture for the design of modular, distributed, and real-time systems. Within the architecture, RTP comprises a substantial Java implementation work performed within the architectural ideas. Since RTP may help in building dynamic and self-adaptive systems, it is based upon reflective mechanisms. Computational reflection, introduced by Smith, is defined as the activity performed by an agent when doing computations about itself [9]. Whereas, architectural reflection [3] is the computation performed by a system about its own internal architecture. We are interested in the second kind of reflection: it reifies architectural features as meta-objects which can be observed and controlled at runtime. The application of architectural reflection helps bringing visibility over the computation performed by the overall system components at the programming level.

In RTP we exploit, as well architectural reflection, a new kind of reflection termed *temporal reflection* [5] enabling the system to reify, observe, and control its own temporal behaviour. RTP, exploiting reflection principles, raises at the application programming level:

– strategies definition (action choice on behalf of events);
– timing issues management (speed-up/slow-down tuning);
– component behaviours definition (adding/removing performable commands);
– system topology definition (adding/removing connected components).

According to the definitions in [8] and in [4], RTP software architecture is defined in terms of components and connectors. RTP extends the above definitions by adding *strategy* as an architectural component. In detail, RTP is based on the following key concepts:

– a system is made up by computational components (components);
– computational components exchange information via alignment components (connectors);
– both computational and alignment components are activated and controlled by supervising components (strategy).

The above three key concepts have lead to individuate three well distinct corresponding roles inside a system: the role of *computing data*, the role of *distributing information*, and the role of *activating both computation and distribution*. RTP assigns these roles to three specific components: *Performer*, *Projector*, and *Strategist*. Performer is the entity expressly designed to *perform* elaboration on own data, Projector is the entity expressly designed to *project* (distribute) data between Performers, and Strategist is the entity expressly designed to device or to employ plans or *stratagems* toward goals like the activation of Performer computations and Projector alignments. The identification of these well-distinguished components emphasises the "separation of concerns" between information processing, information alignment, and their activation. To achieve reuse both off-line and on-line, the following requirements must hold:
Performer should be unaware about both the surrounding environment (in terms of topology) and the system behaviour; moreover it should perform its activities only upon triggered commands;
Projector should be aware about only the Performers it projects, but it should be unaware about the system behaviour; moreover it should perform alignment activities only upon triggered commands;
Strategist should be aware about the overall system behaviour. Strategies should be dynamically created thanks to reflective mechanisms.

Strategies definition implies the definition of planned actions. An action specifies the activity that may be performed by a Performer or a Projector. An action must be fulfilled with temporal constraints, i.e., the time interval in which it should be executed. The Strategist plans system temporal behaviour by adding temporal actions upon a timeline. In this view, a timeline models system temporal behaviour. It is modelled as a monotonic sequence of time instants characterised by the non-decreasing `now()` value of the current time. The current time splits the timeline into a past and a future timeline. The Strategists puts future actions in the future timeline only. When an action is performed, it is moved in the past timeline and it is enriched with additional information about when (actual time) it has been performed.

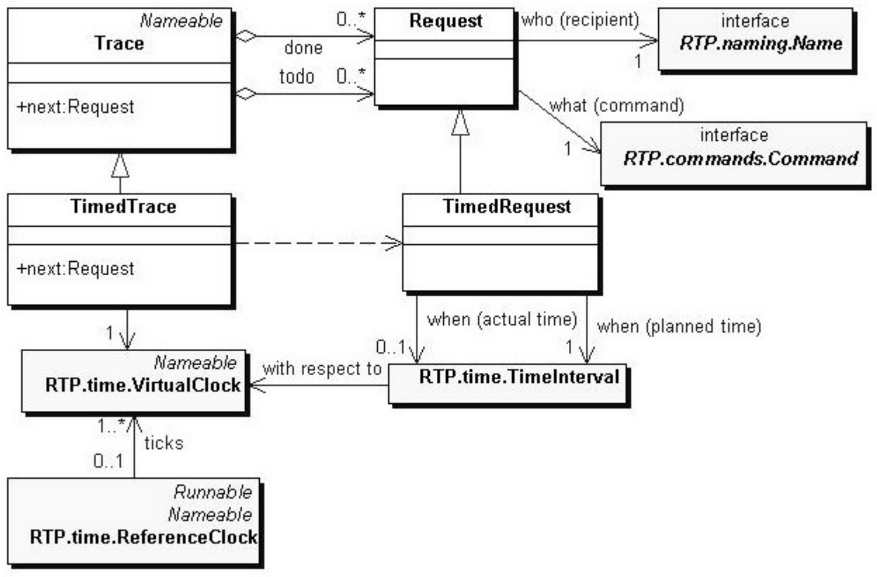

Fig. 1. `TimedRequest` and `TimedTrace`

In terms of RTP concrete architecture (see Figure 1), a Strategist defines system behaviour by manipulating `TimedTraces`. A `TimedTrace` defines system behaviour in terms of partially ordered set of `TimedRequests`. A `TimedRequest` models an action: it is defined as (recipient, command_to_perform, planned_time), where the *recipient* is a Performer or a Projector, *command_to_perform* is the command that the recipient has to execute, and *planned_time* is the time interval in which the command must be delivered and executed (labelled planned). Referring to Figure 1, the other association with `TimeInterval` class (labelled *when (actual time)*) describes the *actual* (if any) temporal interval. This association specifies the time in which the request has been delivered and then executed. The cardinality of the association is 0..1 because it may occur that a request cannot be executed since its planned time is expired with respect to the `now` value. A `TimedTrace` is aware of the current time by means of the associated `VirtualClock` (an active entity that is in charge of advancing current time). When its `next()` method is invoked, it finds for the `TimedRequest` whose planned `TimeInterval` is near to current time. A `TickedEngine` activates the `TimedTrace` `next()` method. `TickedEngine` class is “ticked” each time the reference real time advances[1].

[1] The reason why we do not associate the `TickedEngine` class to a `VirtualClock` is that an engine may “controls” several virtual clocks whose periods are different.

What is relevant is that the model allows strategy changes dynamically during the lifecycle of the system. In fact, since Strategies can be planned at the application domain level, the system behaviour may be easily changed to respond to the specific requirements by changing, adding, and removing Requests inside a Trace (the future portion). Moreover, a Strategist may even change, at the application level, the dynamics of the system in terms of timing issues, e.g., it may accelerate or decelerate some activities exploiting virtual clocks. Figure 2 sketches the system temporal behaviour controlled by a Strategist as provided by RTP.

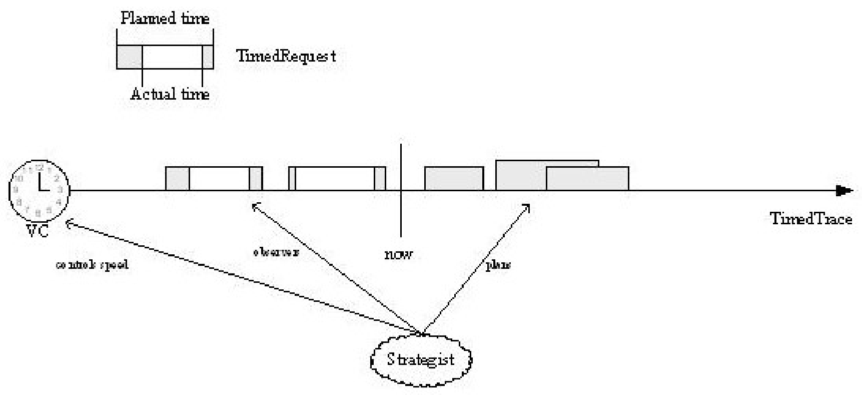

Fig. 2. RTP system temporal behaviour controlled by a Strategist

3 Time related QoS

In general, QoS can be managed by “reflecting” attributes of a system, so that the system can at least monitor itself and possibly control its own performance [1]. A reflected attribute can be read to examine the status of a system component and may also be written (if *causally connected*) to fine tune component behaviour. Of course, system domain affects the set of reflected attributes and their operativity (i.e., read/write ability).

The original version of RTP allows a basic level of QoS management. In fact, a Strategist may analyse delays (e.g., average actual action endings with respect to planned endings) in the past segment of any trace to decide rearrangements in the future segment. This can be done by changing time intervals (i.e., changing planning), thus adapting system behaviour to environment/component conditions. For example, consider a system with one Performer accepting just one kind of command ("snap") and with a plan such the following one[2]:

- **PAST**
- (17:00:00 - 17:01:00 ; 17:00:10 - 17:00:20) [webcam1,snap]
- (17:01:00 - 17:02:00 ; 17:01:15 - 17:01:28) [webcam1,snap]
- (17:02:00 - 17:03:00 ; 17:02:20 - 17:02:32) [webcam1,snap]
- **FUTURE**
- (17:03:00 - 17:04:00 ; nil - nil) [webcam1,snap]
- (17:04:00 - 17:05:00 ; nil - nil) [webcam1,snap]
- (17:05:00 - 17:06:00 ; nil - nil) [webcam1,snap]
- ...

The reader may notice that no request is executed out of time, so no tuning seems necessary. Let now imagine a slightly more complex system, with an added "image compressor" component (Performer) and a plan like this:

- **PAST**
- (17:00:00 - 17:00:30 ; 17:00:10 - 17:00:20) [webcam1,snap]
- (17:00:30 - 17:01:00 ; 17:00:40 - 17:01:05) [jpeg1,pack]
- (17:01:00 - 17:01:30 ; 17:01:15 - 17:01:29) [webcam1,snap]
- (17:01:30 - 17:02:00 ; 17:01:45 - 17:02:03) [jpeg1,pack]
- (17:02:00 - 17:02:30 ; 17:02:20 - 17:02:30) [webcam1,snap]
- (17:02:30 - 17:03:00 ; 17:02:48 - 17:03:02) [jpeg1,pack]
- **FUTURE**
- (17:03:00 - 17:03:30 ; nil - nil) [webcam1,snap]
- (17:03:30 - 17:04:00 ; nil - nil) [jpeg1,pack]
- (17:04:00 - 17:04:30 ; nil - nil) [webcam1,snap]
- (17:04:30 - 17:05:00 ; nil - nil) [jpeg1,pack]
- (17:05:00 - 17:05:30 ; nil - nil) [webcam1,snap]
- (17:05:30 - 17:06:00 ; nil - nil) [jpeg1,pack]
- ...

In this case we may notice that an almost systematic end-of-action delay is born by the *jpeg1* component: 5 seconds, 3 seconds and 2 seconds for the three requests in past trace. We may argue that these delays can be caused by external factors, such as scene changes in front of the webcam, possibly stressing the jpeg compressor with increased image details. Please note also that this delay may not necessarily imply any system inconsistency at the functional level for this example system, but it is an out-of-specification behaviour and it must be treated accordingly. Some kind of corrective strategic action is needed. If no particular high-level requirements are in place (such as "must take a picture every XX seconds") we may think about rewriting the future plan with less

[2] Requests format is "(Planned Begin - Planned End; Actual Begin - Actual End) [recipient, command]". Moreover, Projector details will omitted in every example since they are redundant for the purpose of this paper. Please also note that timings are purely exemplificative.

stringent timings by spreading time intervals to avoid future delaying of actions. We are changing system behaviour of course, and we can only do it if we can rewrite future segments of the plan without breaking other requirements. But the most important remark here is that the system is able to monitor its own temporal behaviour and it can do something about it, or just notify that "the current plan cannot be done".

3.1 Time related QoS extensions

To exemplify extensions added to RTP, here the reader may find a brief list of methods that were introduced to add self monitoring ability, they were added to the `TimedTrace` class to help analysing past trace:

getMissed() returns missed requests;
howManyUnfulfilled() returns the number of unfulfilled requests (it will be overwritten in the `QoSTrace` extension);
getActualDurationLastRequest() returns last request actual duration;
getPlannedDurationLastRequest() returns last request planned duration;
aveActualDuration() returns the average of actual durations in the past;
minBeginDelay() returns the minimum among beginning-of-actions delay;
maxBeginDelay() returns the maximum among beginning-of-actions delay;
aveBeginDelay() returns the average beginning-of-actions delay;
minEndDelay() returns the minimum among ending-of-actions delay;
maxEndDelay() returns the maximum among ending-of-actions delay;
aveEndDelay() returns the average ending-of-actions delay;
maxTimeInterval() returns the maximum time interval set for any action in the past trace;
minTimeInterval() returns the minimum time interval set for any action in the past trace (this can be useful to check the "quantum" time interval for a system, if the corresponding action was executed on time).

Every method listed above has different versions: without parameters and with "selective" parameters. Selective parameters are needed when statistics need to be computed on subsets of past history. For example, if we need to know the average end delay of every "pack" action, instead of the same average but generically computed on every action in the past, we may use an `aveEndDelay(Filter)` method.

4 RTP extensions for resource QoS

This section shows other extensions to let the system take into account factors not only related to time, but also to generic allocation of resources. In our extended RTP framework, resource QoS is managed at two different levels:

At Plan level it is possible to tweak system behaviour by rearranging requests in terms of time, order, duration, and **budgeted resources** in the future part of the trace. Hints for correct tuning come from the analysis of past history, by calculating values such as "average begin delay" or "average memory consumption" of actions and so on;

At Performer level it is possible to monitor resource consumption of QoS-enabled Performers; some kind of control is also possible since every QoS-enabled Performer will not accept a request that is clearly impossible to fulfill with current resources (compared to the budgeted ones).

At Plan level we need some kind of resource allocation information, the same as it is for time information. Moreover, we need to keep track of planned resources (budget, forecasts) and of actual usage/consumption. The QoS RTP extensions model resource allocation by associating a set of resource budgets to every request. Every request, if and when honored, is then piggybacked with information about the actual resource consumption needed for its execution. This "enriched" request can be studied in terms of system usage. For example, let us augment the above "webcam+compressor" system with resource data and imagine a plan such as the following[3]:

– **PAST**
– (17:00:00 - 17:00:30 ; 17:00:10 - 17:00:20) {ram:3k,2k} [webcam1,snap]
– (17:00:30 - 17:01:00 ; 17:00:40 - 17:01:05) {ram:10k,5k} [jpeg1,pack]
– (17:01:00 - 17:01:30 ; 17:01:15 - 17:01:29) {ram:3k,2k} [webcam1,snap]
– (17:01:30 - 17:02:00 ; 17:01:45 - 17:02:03) {ram:10k,8k} [jpeg1,pack]
– (17:02:00 - 17:02:30 ; 17:02:20 - 17:02:30) {ram:3k,2k} [webcam1,snap]
– (17:02:30 - 17:03:00 ; 17:02:48 - 17:03:02) {ram:10k,11k} [jpeg1,pack]
– **FUTURE**
– (17:03:00 - 17:03:30 ; nil - nil) {ram:3k,nil} [webcam1,snap]
– (17:03:30 - 17:04:00 ; nil - nil) {ram:10k,nil} [jpeg1,pack]
– (17:04:00 - 17:04:30 ; nil - nil) {ram:3k,nil} [webcam1,snap]
– (17:04:30 - 17:05:00 ; nil - nil) {ram:10k,nil} [jpeg1,pack]
– (17:05:00 - 17:05:30 ; nil - nil) {ram:3k,nil} [webcam1,snap]
– (17:05:30 - 17:06:00 ; nil - nil) {ram:10k,nil} [jpeg1,pack]
– ...

This example is time-wise identical to the one shown before, but in this case every request has an associated "resource budget" information. The reader may notice that the past trace contains a request that was executed without complying to the "ram" budget that was set for that request. This is an example of a request without *strict* budget: these requests are honored even when the Performer realises that the budget will not be satisfied. QoS RTP extensions allow also the definition of strict budgets. Performers must not execute strict-budget

[3] The new request format is "(Planned Begin - Planned End; Actual Begin - Actual End) **{resource: planned, actual}*** [recipient, command]". The "{}*" syntax stands for: "{} a number of times".

requests if they cannot guarantee budget satisfaction. This example shows a situation for which there is no straightforward solution of course: there is no simple way of rearranging requests to bind memory consumption of a component, but, again, the system has at least a way to know about the problem.

4.1 Resource QoS extensions

Figure 3 shows the new `QoSRequest` class, an extension of `TimedRequest`, to keep track of information about planned and actually consumed resources. A `Resource` instance is a representation of a physical resource. As an example, an instance of `Memory (extends Resource)` is used to represent the amount of memory (e.g. 10KB) that must not be exceeded to perform a single action.

In this version of RTP, `Performer` duty has been loaded with an additional responsibility: filling consumption information into any request processed. An Engine gets a Request from its associated Trace and dispatches it to a Performer. The receiving Performer examines the Resource budget associated to the Request to decide if it may be fulfilled, and if so it executes the Request. After execution, the Performer gathers consumption info (self generated) and fills it into the Request for later analysis. The Request goes back to the the trace, in the past segment. Now the trace is very rich in terms of meta-information about past system behaviour. Requests in the past segment are associated with time and resource-consumption information, thus new methods to extract statistics were added to the `QoSTrace` (see Figure 3), such as:

minResourceUsage(Resource) returns minimum resource usage for a particular Resource;
maxResourceUsage(Resource) returns maximum resource usage for a particular Resource;
aveResourceUsage(Resource) returns average resource usage for a particular Resource;
howManyUnfulfilled() (overridden) returns the number of unfulfilled requests in the past, not only in terms of time (TimedTrace version) but also in terms of resources;
***xxx*ResourceUsage(Resource, Filter)** returns (min|max|ave) resource usage for a particular Resource, selective versions (not shown).

At Performer level, newly added RTP components are the `QoSPerformer` and the `QoSProjector`. They offer the capability to keep track of their own performance by recording "statistics". A set of continuously updated `Statistics` instances is associated to every `QoS(Performer|Projector)` to make these components aware of their own past performance. The new `QoS(Performer|Projector)` operates a double information fill: one on the `QoSRequest` and another on itself by updating consumption info on its own set of `Statistics` instances. This way, the longer a system runs, the "wiser" it becomes in terms of the ability to decide if an action can be executed within budget, since every node accumulates information about its own performance-wise behaviour. If the `RespectResourcePlanning`

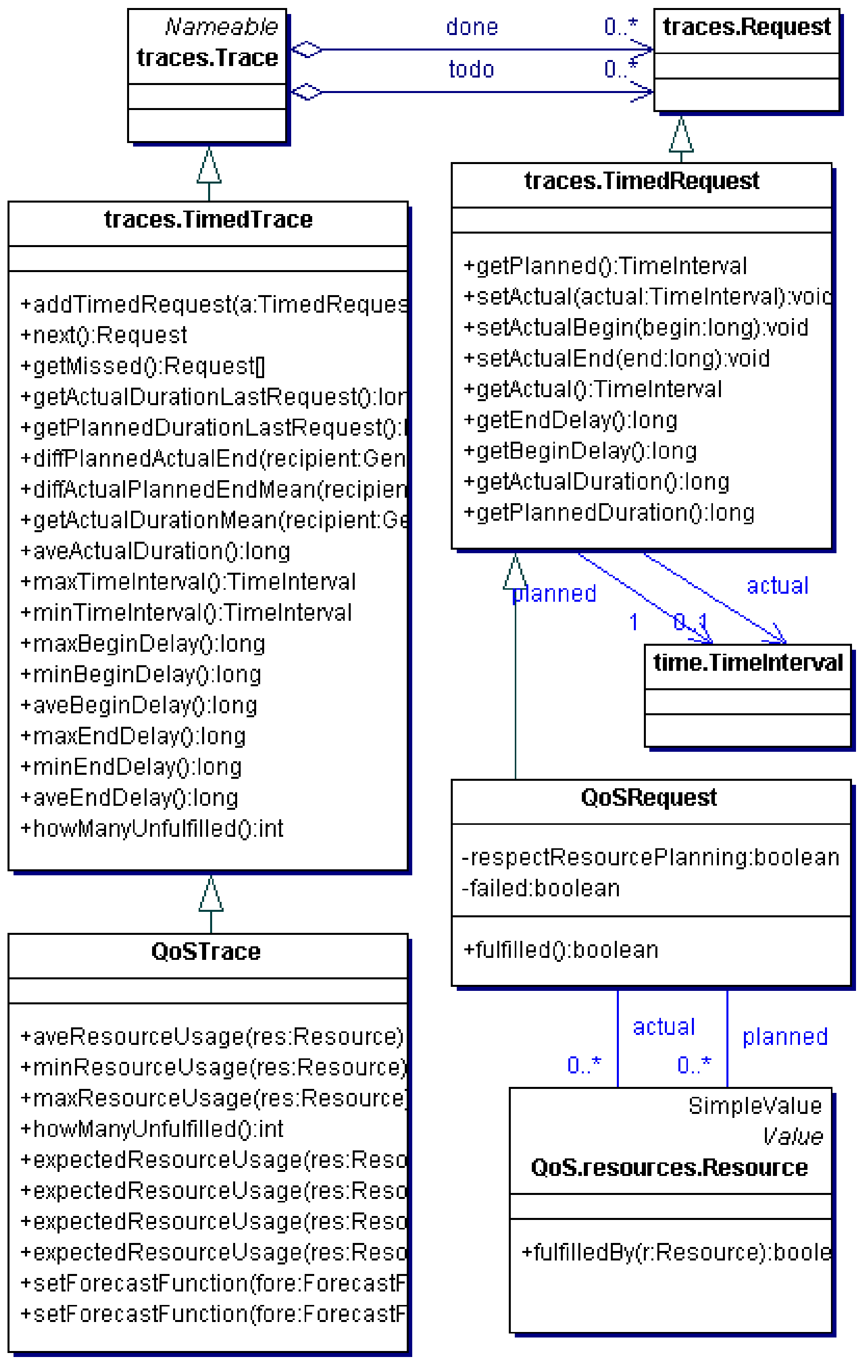
Nameable
traces.Trace
done
0..*
todo
0..*
traces.Request
traces.TimedTrace
+addTimedRequest(a:TimedReques
+next():Request
+getMissed():Request[]
+getActualDurationLastRequest():lor
+getPlannedDurationLastRequest():l
+diffPlannedActualEnd(recipient:Gen
+diffActualPlannedEndMean(recipien
+getActualDurationMean(recipient:Ge
+aveActualDuration():long
+maxTimeInterval():TimeInterval
+minTimeInterval():TimeInterval
+maxBeginDelay():long
+minBeginDelay():long
+aveBeginDelay():long
+maxEndDelay():long
+minEndDelay():long
+aveEndDelay():long
+howManyUnfulfilled():int
traces.TimedRequest
+getPlanned():TimeInterval
+setActual(actual:TimeInterval):voic
+setActualBegin(begin:long):void
+setActualEnd(end:long):void
+getActual():TimeInterval
+getEndDelay():long
+getBeginDelay():long
+getActualDuration():long
+getPlannedDuration():long
planned
actual
1
0..1
time.TimeInterval
QoSRequest
-respectResourcePlanning:boolean
-failed:boolean
+fulfilled():boolean
actual
planned
0..*
0..*
QoSTrace
+aveResourceUsage(res:Resource)
+minResourceUsage(res:Resource)
+maxResourceUsage(res:Resource)
+howManyUnfulfilled():int
+expectedResourceUsage(res:Reso
+expectedResourceUsage(res:Reso
+expectedResourceUsage(res:Reso
+expectedResourceUsage(res:Reso
+setForecastFunction(fore:ForecastF
+setForecastFunction(fore:ForecastF
SimpleValue
Value
QoS.resources.Resource
+fulfilledBy(r:Resource):boole

Fig. 3. QoS-enabled class hierarchy

attribute of a request has been set, the receiving Performer should not honor that request unless absolutely sure about respecting planned resource budget. This mechanism is not an easy task to implement (it may even be undecidable). At the moment, the only information taken into account by the receiving Performer is its own set of statistics. By examining its own past performance, a Performer may have a clue on the possibility to respect a budget. The boolean `fulfilled()` method on the `QoSRequest` returns false on a yet-to-be-executed request and true for an executed request that has respected its budget set. `QoSRequest fulfilled(Resource)` and `Resource fulfilledBy(Resource)` methods are utilities to compare planned budgets and actual consumptions.

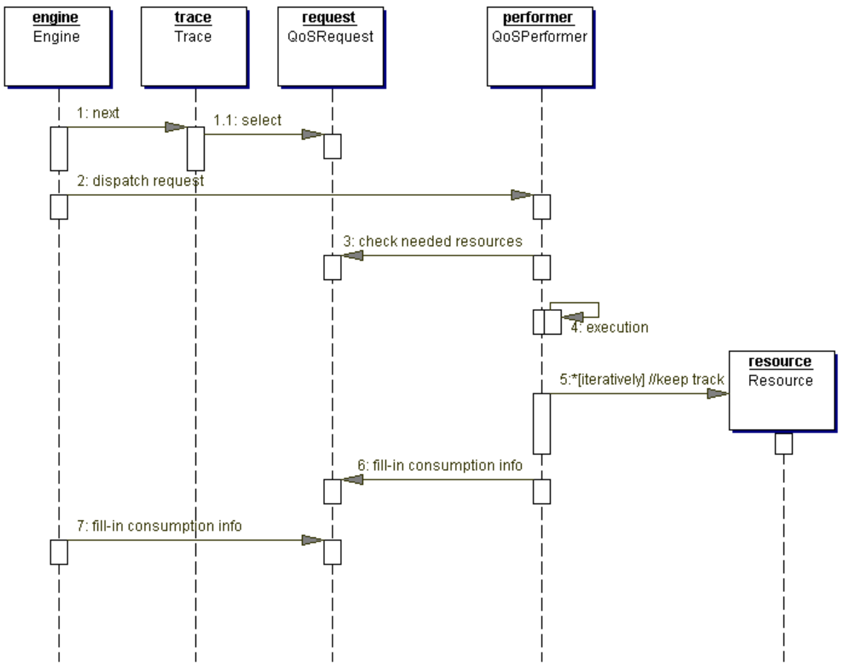

Fig. 4. QoS traces sequence

5 Conclusion

This paper presents RTP extensions to manage QoS parameters for time-related and resource consumption aspects. These extensions were added to keep track (and take control) of timely behaviour and resource consumption at every logical level supported by the framework itself. QoS can now be managed (monitored and controlled) by operating at different levels:

Plan level monitoring, plans can now be queried to know about past system performance in terms of timely behaviour and resource consumption;

Node level monitoring, every Performer exports information about its own behaviour in terms of statistics on past performance;

Strategic control, active system behaviour management can be done by changing the future part of the trace (i.e., replanning), this is no different than before, but in this new version of RTP strategies can take into account more time related aspects and the newly added resource consumption information spread throughout the framework.

References

1. G. Blair, A. Amdersen, L. Blair, and G. Coulson. The role of reflection in supporting dynamic qos management functions. In *Proceedings of the IEEE/IFIP International Workshop on Quality of Service (IWQoS'99)*, 1999.
2. Jorge Cardoso, Amit Sheth, and John Miller. Workflow quality of service. *International Conference on Enterprise Integration and Modeling Technology and International Enterprise Modeling Conference*, 2002.
3. W. Cazzola, A. Savigni, A. Sosio, and F. Tisato. Architectural Reflection: Bridging the Gap Between a Running System and its Architectural Specification. In *Proceedings of the 2nd Euromicro Conference on Software Maintenance and Reengineering and 6th Reengineering Forum*, March 1998.
4. A.H. Eden and R. Kazman. Architecture, design, implementation. In *Proceedings of the $25^t h$ IEEE International Conference on Software Engineering*, pages 149–159, May 2003.
5. Daniela Micucci, Sergio Ruocco, Francesco Tisato, and Andrea Trentini. Time sensitive architectures: a reflective approach. *International Symposium on Object Oriented Real-time distributed Computing (ISORC)*, 2004.
6. G.S. Carrapatoso E. Moreira, R.S. Blair. A reflective component-based and architecture aware framework to manage architecture composition. In *Distributed Objects and Applications, 2001. DOA '01. Proceedings*, 2001.
7. OMG. Uml profile for schedulability, performance and time. http://www.omg.org/cgi-bin/doc?formal/2003-09-01 (URL last visited Oct, 1 2004), September 2003.
8. M. Shaw and D. Garlan. *Software Architecture. Perspective on an Emerging Discipline.* Prentice Hall, 1996.
9. Brian Cantwell Smith. Reflection and semantics in a procedural language. Technical Report 272, MIT Laboratory of Computer Science, 1982.