Scheduling Real-Time Components Using Jitter-Constrained Streams

Claude-Joachim Hamann Steffen Zschaler
Fakultät Informatik
Technische Universität Dresden
Dresden, Germany
E-mail: {claude-joachim.hamann,steffen.zschaler}@tu-dresden.de

Abstract

Component-based applications require good middleware support. In particular, business logic should be separated from management code for guaranteeing nonfunctional properties of a system. We present an approach called Container-Managed Quality Assurance, *in which a component container uses nonfunctional specifications of components to determine how to use these components, and which system resources to allocate, to provide certain services with guaranteed nonfunctional properties.*

As an example, we show how this technique can be applied to automatically allocating CPU and memory resources for components with real-time constraints. To this end, we use a mathematical model based on jitter-constrained streams, a mathematical abstraction of event streams.

Keywords: Component-Based Software, Real Time, Scheduling, Container-Managed Quality Assurance

1. Introduction

Component-based software engineering (CBSE) [13] has become an important approach to software development, because it promises more efficient, faster, and less error-prone development cycles. This promise is based on the idea of reusing pre-fabricated and individually tested software components provided by third-party developers. In this scenario, component developers focus on a specific business area where they develop great experience and expertise and are, thus, able to produce high-quality software components for this business area. These components are then bought by other parties who integrate them to produce customer-specific applications. Of course, this implies that the components will be reused in a multitude of contexts, not all of which can be predicted by the original developer.

This is especially relevant when discussing non-functional properties of components (in particular Quality-of-Service properties) or the resulting application. These properties are strongly affected by the usage context [17, 18]. It is, therefore, necessary to separate the concerns of business logic and support for nonfunctional properties of the resulting application. Note that for the functional side this has already been done. Components are typically executed in a component infrastructure (also called *container*), which—among other things—provides services for finding and invoking the services of other components. This leads us to the notion of *Container-Managed Quality Assurance:* Component developers provide specifications of the intrinsic[1] nonfunctional properties of a component. This information is used by the container, together with information about the execution context (available resources, parameters of the request stream, etc.) to manage the components in such a way that the services provided by the system as a whole have certain (specified) extrinsic nonfunctional properties. In essence, container-managed quality assurance serves to automate the implementation of scheduling and resource allocation decisions based on declarative descriptions of the components in an application.

This paper is structured as follows. In the next section, we explain our idea of container-managed quality assurance and how support for guaranteeing response time of a service can be implemented. Section 3 then discusses jitter-constrained streams—the mathematical foundation of our approach—and presents some theoretical results. Finally, Sect. 4 uses these approaches to complete the container presented in Sect. 2. The paper concludes with a review of related work.

2. Container-Managed Quality Assurance

Figure 1 shows the basic structure of a system for container-managed quality assurance as a Unified Mod-

[1] cf. [18] for a more detailed discussion of this terminology.

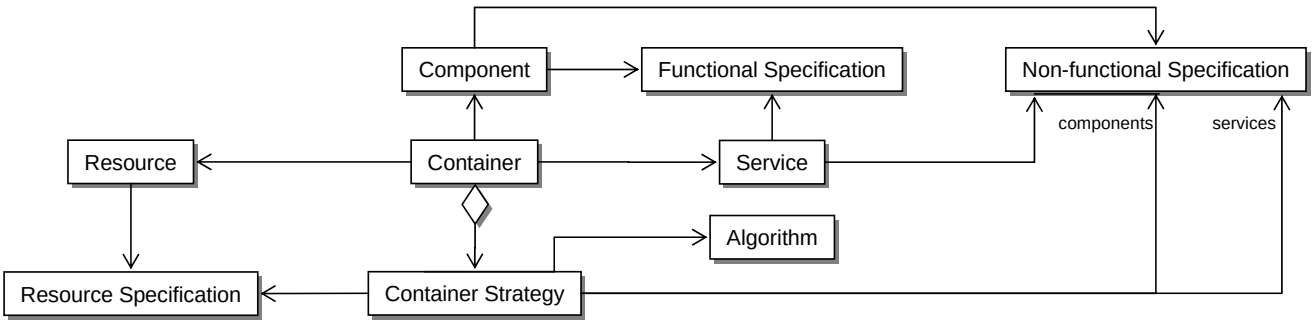

Figure 1. UML model of a system using container-managed quality assurance

elling Language (UML) [10] class diagram. In the centre of the diagram, we see the `Container` that manages `Components` and uses them to provide `Services`. Obviously, both components and services have a functional and a nonfunctional specification. The container implements container-managed quality assurance through so-called `Container Strategies`—`Algorithms` for transforming components with a certain intrinsic non-functional specification and available `Resources` (as per a `Resource Specification`) into services with a certain extrinsic nonfunctional specification.

In this paper, we discuss the mathematical background and approach for one such container strategy. A component A provides one operation of which the execution time is known. The container will receive a stream of requests for the functionality provided through this operation. Requests occur periodically, but there may be a certain jitter. The container strategy must reserve sufficient resources (in particular, CPU cycles and, potentially, buffer space for incoming requests) and create sufficiently many instances of A (perhaps running on different processors of a multi-processor machine) to honour these requests and guarantee a certain upper bound for the response time of each request.

Such a container strategy consists of two parts:

1. The container strategy must compute the required buffer size, and the number of component instances required. For each component instance, the container strategy allocates the required amount of memory and starts a worker thread that will manage invocations of operations offered by the component. For each thread it allocates a task with the CPU scheduler. The execution time of the task is derived from the execution time of the component's operation. Period and deadline are determined such as to guarantee the required response time. Additionally, the container strategy allocates a buffer for incoming requests. The resulting system is depicted in Fig. 2.

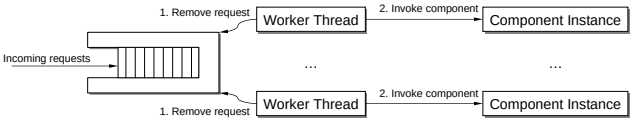

Figure 2. Dynamic structure of a component managed by our container strategy

2. The worker thread executes an infinite loop, in which it first blocks until the scheduler signals the beginning of a new period. It then tries to remove a request from the buffer, and if there is one available, it invokes the component's operation with the new request. In this way, all knowledge about how to communicate with the underlying CPU scheduler is encapsulated in the container. In fact, the component does not need to know that there will be any scheduling at all.

The worker thread can be represented in pseudo-code as follows:

```
while (true) {
  cpuScheduler.waitForNextPeriod();

  Request r = bufferManager
                .getNextRequest();

  if (r != NULL) {
    bufferManager.sendResult (
        invoke (componentInstance, r));
  }
}
```

where `cpuScheduler` represents an interface for communicating with the underlying resource manager and `bufferManager` represents an interface of the container for accessing the buffer and sending back results to the correct client.

In order for this approach to work, the container strategy needs a way to determine the required buffer size and

the scheduling parameters (number of component instances, period and execution time of the underlying tasks) from its knowledge about worst-case execution time, required response time, and request-stream parameters. The following two sections deal with this issue: The next section provides the mathematical foundations, while Sect. 4 explains how this can be applied for our container strategy.

3. Jitter-Constrained Streams (JCSs)

The formalism "jitter-constrained periodic stream" presented in [5, 6] allows the treatment of sequences of events that normally occur with a constant period, but can be too early or too late within certain defined boundaries. We use this model to determine buffer sizes and delays. In the following, we modify the definitions given in [5] and summarise some results. Then, we apply the formalism to the problem described in Sect. 1.

[5] discusses streams of events (interpreted as sending or receiving of data packets of constant size) of the following type: Starting with a time t_0, events occur sequentially with a constant temporal distance of T. However, the occurrence of events may deviate from this constant schedule. In particular, events may occur too early by a maximum time τ or too late by a maximum time τ'. The time between the occurrence of two events must be at least D. T is, thus, the average time between two consecutive events. [5] sets $t_0 = 0$ immediately and assumes $0 < D < T$. These parameters T, D, τ, τ' are then used to derive—among others—burst length and buffer sizes.

Further studies [2, 7] require the discussion of cases other than $t_0 = 0$. Additionally, this enables us to completely represent jitter by only *one* parameter as shown in [6, Corollary 1]. Both theory and application remain the same whether this parameter represents delay or early occurrence. We, therefore, decided to interpret it as delay, as this avoids negative times when studying streams in isolation. Moreover, [6] discusses the border-line cases $D = 0$ and $D = T$ as additional extensions of the original definition. If we identify events with the time of their occurrence a_i and abstract from the concrete notion of *time*, the following modification of the original definition suggests itself:

Definition 1 (Jitter-constrained periodic stream) Let

$$D, T, \tau, t_0 \in \mathbb{R} \quad \text{with} \quad 0 \le D \le T,\ T > 0,\ \tau \ge 0. \quad (1)$$

A *jitter-constrained periodic stream* with average value T, minimum value D, jitter τ and initial value t_0 (jitter-constrained stream, JCS) is a sequence $(a_i)_{i=0,1,\ldots}$ of numbers for which the following conditions hold:

$$\left.\begin{array}{rcl} a_i & \in & [t_0 + iT, t_0 + iT + \tau] \subseteq \mathbb{R} \\ a_{i+1} - a_i & \ge & D \end{array}\right\} \forall i \in \mathbb{N}. \quad (2)$$ ■

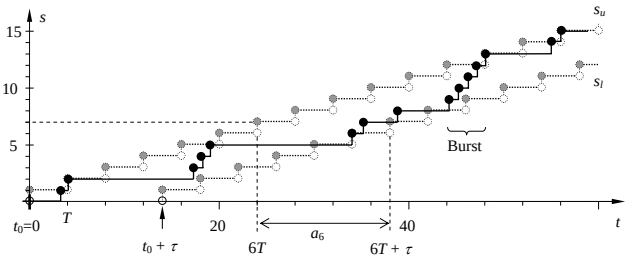

Figure 3. Limits ····· and trajectory —— of a JCS $S = (4, 1, 14, 0)$

In the following, we are going to interpret all such streams as temporal sequences of events. Therefore, we keep using the terms *period*, *minimum distance*, *jitter* (or delay), and *start time* for the parameters. Symbolically we write $S = (T, D, \tau, t_0)$. This does not denote a concrete stream, but the extent of all event sequences for which Def. 1 holds with the given parameter values. A concrete stream $s = (a_i)_{i=0,1,\ldots}$ is then given through the concrete time instants a_i at which the individual events occur. The border-line cases for S are the two concrete streams s_u and s_l[2], for which all events occur at the earliest and latest time instant, respectively.

From this last point of view, we can understand stream s as a trajectory of the set of event sequences defined by $S = (T, D, \tau, t_0)$ corresponding to Def. 1. This introduces a view of a JCS deviating from [5]: A concrete stream is represented as a (right-continuous) step function $s(t)$ with salti a_i and step height 1, where $s(t)$ is the number of events that have occurred until time t. Figure 3 shows the two streams s_u and s_l as well as the interval for a_6 (of the 7th event) for the stream $S = (T = 4, D = 1, \tau = 14, t_0 = 0)$. It is obvious that:

$$s_u(t) = \begin{cases} 1 + \left\lfloor \frac{t - t_0}{T} \right\rfloor & t \ge t_0 \\ 0 & t < t_0 \end{cases}$$
$$s_l(t) = \begin{cases} 1 + \left\lfloor \frac{t - \tau - t_0}{T} \right\rfloor & t \ge t_0 + \tau \\ 0 & t < t_0 + \tau \end{cases} \quad (3)$$

Every concrete stream s is then a step function $s(t)$ bounded by s_u from above and s_l from below and where D is the minimum width of a step and $\tau + T$ is the maximum step width (cf. Fig. 3).

It turns out, that for all analytic studies of such streams one type of event sequences is particularly important: Sequences of maximum length where all events occur with minimum distance—called bursts (cf. Fig. 3). From [5], we

[2] For **u**pper and **l**ower bound.

know that the burst length L for a JCS $S = (T, D, \tau, t_0)$ is

$$L = 1 + \left\lfloor \frac{\tau}{T - D} \right\rfloor \tag{4}$$

and the earliest starting time b_e for a burst is

$$b_e = t_0 + (L - 1)(T - D). \tag{5}$$

The main focus of the discussion in [2, 5, 7] is a buffer P that is being filled by a JCS $S = (T, D, \tau, t_0)$ with data packets of equal size. Data packets are removed from the buffer with a constant temporal distance of T. Two special types of streams are studied and for both the minimum buffer size p for lossless processing is determined. Here, we generalise these results to arbitrary JCS and, additionally, we compute the maximum time a data packet remains in the buffer (wait time t_w in terms of queueing theory). We call the filling process the *producer process* PP and the removing process the *consumer process* CP.

As in [5] we study the following removal strategy, which we will call *undelayed*:

1. If a data packet arrives in an empty system it is removed immediately.
2. When the consumer has removed a packet, it is ready to remove the next packet after the constant time T.
3. The consumer pauses if no data packets are available for processing. In particular, the consumer only starts when the first data packet arrives.
4. When a data packet is removed from the buffer, the corresponding slot in the buffer is available immediately. This means that an arriving data packet does not require additional buffer space if the consumer is ready for a removal at the time of arrival.
5. Packets are removed in FIFO order.

Then the following holds:

Theorem 1 A producer process PP produces data packets in the form of a JCS $S = (T, D, \tau, t_0)$. A consumer process CP consumes these packets undelayed with the constant time T. Then, a buffer of size

$$p = \left\lceil \frac{\tau}{T} \right\rceil \tag{6}$$

is sufficient for lossless processing. The maximum wait time t_w of a data packet in the buffer is

$$t_w = \tau. \tag{7}$$ □

The proof of this theorem can be found in the appendix.

In the following, we apply this formalism to two specialised producer–consumer problems from the point of view of a multi-processor system to provide a solution to the task discussed in Sect. 1. More precisely, a server process receives requests from a client process. The requests arrive as a JCS, but with an average rate that is higher than what the server can handle. Processing time is constant and requests are processed undelayed (as defined above). Incoming requests are placed in a buffer (unless the server is available for processing the request immediately), the size of the request is not relevant. To ensure stationary behaviour of the system, the server process must, thus, be available in more than one instance. We are interested in computing the required minimum number of instances, the required buffer size, and the maximum wait time of a request for a buffer shared by all instances.

We use the following terminology:

$C = (T, D, \tau, 0)$ Input stream (a JCS) produced by the client process

$\tilde{T} > T$ Processing time of the server process for one request. This is, thus, the constant time after which the server can first remove a new request from the buffer.

n Number of required instances of the server process running in parallel

p Minimum buffer size required for lossless processing

t_w Maximum wait time of a request.

The actual starting time of C and the precise description of the consumer process (server) as a JCS are not relevant in this context. We, initially, study the removal strategy where the server works undelayed in the sense defined above.

For the number of required instances $I_1 \dots I_n$ obviously

$$n = \left\lceil \frac{\tilde{T}}{T} \right\rceil. \tag{8}$$

A strict limit for the buffer size p of a common buffer P, which stores all incoming events sequentially and is emptied round-robin style by the individual instances can be found by noting that s_l represents the time instants when a data packet is being removed from the buffer (cf. Fig. 4). Hence, we can reuse the argument from Theorem 1 and get: For lossless processing it is sufficient to use a buffer of size $\lceil \tau/T \rceil$. Figure 4 additionally shows that the maximum wait time of a request in the buffer is equal to τ. Furthermore, we can see from the figure that the two limits (buffer size and wait time) are strict for request streams of the following form: n requests occur individually each at the latest possible time. After that, as soon as possible, a burst arrives.

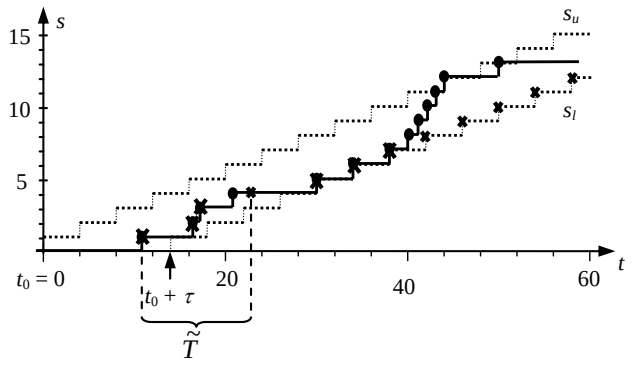

Figure 4. Removal times ✖ for a JCS $S = (4, 1, 14, 0)$ and a sufficiently large buffer. Elements are removed with $\tilde{T} = 12$

Should—differently from Fig. 4—$\tilde{T}$ be no integer multiple of T the maximum wait time reduces by $nT - \tilde{T}$.

To summarise:

Theorem 2 Let P be a buffer filled by a client process CP in the form of a JCS $C = (T, D, \tau, t_0)$ and emptied by n instances of a server process SP. SP removes elements undelayed starting at t_0 and processes them in the constant time $\tilde{T} > T$. Then

$$n = \left\lceil \frac{\tilde{T}}{T} \right\rceil \tag{9}$$

is the minimum number of instances required so that the buffer does not grow beyond all bounds. It is then sufficient to provide a buffer P of size

$$p = \left\lceil \frac{\tau}{T} \right\rceil \tag{10}$$

to guarantee lossless processing of all requests. Furthermore,

$$t_w = \tau + \tilde{T} - nT \tag{11}$$

gives the maximum wait time (maximum delay until a request is removed from the buffer) for an element in buffer P. □

Finally, we discuss a modified server behaviour more closely modelling real-world situations. The server now does not immediately remove a data packet arriving at an empty buffer (as this would require notification of the server process and, thus, induce additional overhead); instead, it checks the buffer in regular intervals. We call the corresponding time instants *sampling points.* In analogy to above, we use the following removal strategy, which we will call *strictly periodic:*

1. The server removes data packets from the buffer at a constant rate with no jitter beginning at a certain start time.
2. If the buffer is empty when the server checks it, the server pauses until the next sampling point.
3. A data packet cannot be removed from the buffer immediately upon arrival, but only at the next possible later sampling point.
4. When a data packet is removed from the buffer, the corresponding slot in the buffer is available immediately.
5. Packets are removed in FIFO order.

We let instances work with a relative offset $T' = \frac{1}{n}\tilde{T}$. This leads to the following formalisation:

As before, let $C = (T, D, \tau, 0)$ be the input stream induced by the client process, and $\tilde{T} > T$ be the constant service time of the server. With

$$T' = \frac{1}{n}\tilde{T} \qquad \text{offset between server instances} \tag{12}$$

the server instances then induce n strictly periodic streams, which as JCSs can be written:

$$V_i = \left(\tilde{T}, \tilde{T}, 0, iT'\right), \quad i = 0, \ldots, n-1.$$

This results in a process V with which data packets can be removed from the buffer and obviously $V = (T', T', 0, 0)$. The actual removal process can be derived from V by "stretching" in the direction of t by T' whenever V does not find a data packet in the buffer. It should be noted that it can be easily seen from (8) that

$$T' = xT \quad \text{with} \quad x \in \left(\frac{n-1}{n}, 1\right].$$

Intuitively, a burst requires the most space in the buffer. The longest wait time occurs, when all server instances are occupied for as long as possible when a bursts arrives. Hence, we will study the following concrete stream $s \in C$ for determining buffer size p and wait time t_w: n events at the latest possible time are followed as soon as possible by a burst:

$$\begin{aligned} a_i &= \tau + iT, \quad i = 0, \ldots, n-1 \\ a_n &= nT + b_e = nT + (L-1)(T-D) \end{aligned}$$

(L and b_e from (4) and (5)).

Because of $T' \leq T$, the events occurring before a_{n-2} will be removed until the next event arrival; also at a_{n-1}, the buffer is empty. We will now discuss the worst-case situations for the cases where the buffer is empty or contains one last element when the burst arrives.

Let

$$\Delta = a_n - a_{n-1}$$

be the distance between the burst and the last event before it (see Fig. 5). Then

$$\Delta \quad = \quad T + (L-1)(T-D) - \tau.$$

If $T' \leq \Delta$ (Fig. 5 a)) a_{n-1} will, thus, be removed at the latest when a_n arrives, so that the burst hits an empty buffer. In the worst case, a_n is a sampling point of V, so that the first event of the burst will not be taken from the buffer immediately. The required buffer size p is then L reduced by the number of sampling points of V during the arrival of the burst, more precisely in the interval $(a_n, a_n + (L-1)D]$:[3]

$$p = L - \left\lfloor \frac{(L-1)D}{T'} \right\rfloor = \left\lceil \frac{L(T'-D)+D}{T'} \right\rceil.$$

If, however, $T' > \Delta$ (Fig. 5 b)) in the worst case, at a_n the buffer contains exactly one element, which arrived at a previous sampling point. Consequently, p is the difference of $L+1$ and the number of sampling points in the interval $(a_{n-1}, a_{n-1} + \Delta + (L-1)D]$:

$$p = L+1-\left\lfloor \frac{\Delta + (L-1)D}{T'} \right\rfloor = \left\lceil \frac{(L+1)T' + \tau - LT}{T'} \right\rceil$$

The maximum wait time t_w can then be derived easily as the distance between the arrival of the last burst event and its removal from the buffer:

$$t_w \quad = \quad \begin{cases} L(T'-D)+D & \text{for } T' \leq \Delta \\ (L+1)T' + \tau - LT & \text{for } T' > \Delta. \end{cases}$$

Packets arriving after the burst cannot increase the buffer, as their distance to the burst will be at least a multiple of $T \geq T'$, so that at least one packet has been removed before their arrival. If the events $a_0, \ldots, a_{n-1}$ arrive earlier than at their respective latest possible moment, they will still be removed at the times just determined (possibly earlier). Hence, the situation cannot be made worse in this way. This argumentation also holds for the complete future stream behaviour.

Finally, the two examples presented in Fig. 5 show that the estimates given are strict, because the easily read-off buffer sizes and wait times coincide with the calculated ones ($p = 4, t_w = 8.5$ and $p = 5, t_w = 13.5$, resp.; in case b) the first burst element is removed at $t = 24.5$). Additionally, it is easy to see (and provable) that the values determined for $T' > \Delta$ are at least as big as for the opposite case. Hence, these values can be taken as a generic (albeit no longer strict) upper bound.

[3] Note: $\forall x \in \mathbb{R} : -\lfloor x \rfloor = \lceil -x \rceil$

We summarise these results in the following theorem, including also the case $T' < D$. Of course, all statements hold also for $\tilde{T} = T$ and $\tilde{T} < T$. This theorem is thus a generalisation of the studies presented above.

Theorem 3 A buffer P is filled by a producer process PP in the form of a JCS $C = (T, D, \tau, t_0)$ and emptied by n instances of a consumer process CP removing elements from the buffer following a strictly periodic removal strategies starting at t_0 and with a constant period of $\tilde{T}$. Then

$$n = \left\lceil \frac{\tilde{T}}{T} \right\rceil$$

is the minimum number of instances required so that P does not grow beyond all bounds. When starting the individual instances with a relative offset

$$T' = \frac{1}{n}\tilde{T} \tag{13}$$

lossless processing can be guaranteed with a buffer of size

$$p = \begin{cases} \left\lceil \frac{(L+1)T' + \tau - LT}{T'} \right\rceil & \text{for } T' > \Delta \\ \left\lceil \frac{L(T'-D)+D}{T'} \right\rceil & \text{for } D \leq T' \leq \Delta \\ 1 & \text{for } T' < D \end{cases} \tag{14}$$

resulting in a wait time

$$t_w = \begin{cases} (L+1)T' + \tau - LT & \text{for } T' > \Delta \\ L(T'-D)+D & \text{for } D \leq T' \leq \Delta \\ T' & \text{for } T' < D \end{cases} \tag{15}$$

where $L = 1 + \left\lfloor \frac{\tau}{T-D} \right\rfloor$ and $\Delta = T + (L-1)(T-D) - \tau$. □

4. Application to Container-Managed Quality Assurance

In the previous section, we discussed a mathematical model for streams of events and derived some results from this model. In this section we are going to apply these results to the realisation of the container strategy from Sect. 2. Our strategy requires the following inputs:

$R = (T, D, \tau, 0)$ the description of the stream of incoming requests for the operation provided by the component formulated as a JCS

t_e the worst-case execution time of the component operation when executed in an empty system

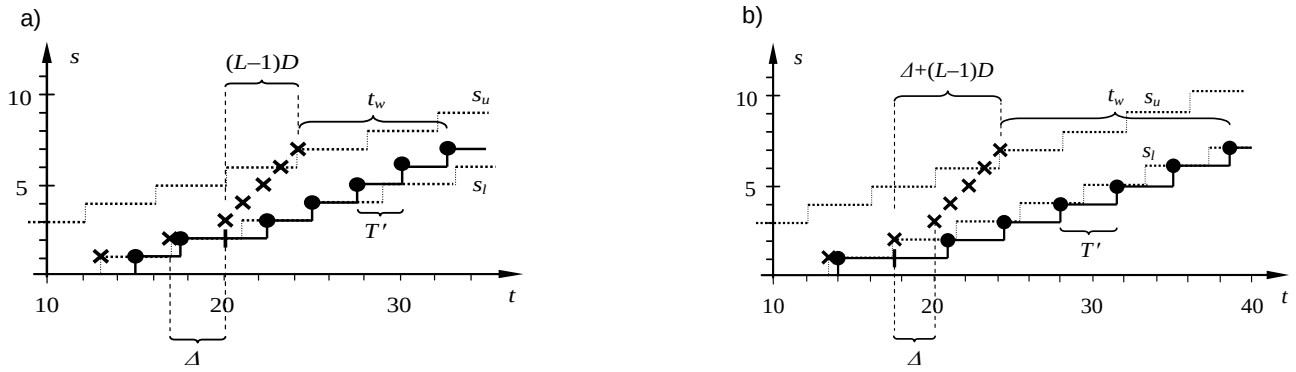

Figure 5. Removal processes for a producer process with $C = (4, \tau, 1, 0)$. I represents a sampling point without removal. a) $\tau = 13, \tilde{T} = 5 \rightarrow 2.5 = T' \leq \Delta = 3$ **b)** $\tau = 13.5, \tilde{T} = 7 \rightarrow 3.5 = T' > \Delta = 2.5$

m the maximum amount of memory required by one incoming request.

The strategy produces the following output (if the task can be solved at all; that is, if sufficient resources are available):

n the number of instances of the component that it will create to process the requests

t_r the response time of the service provided by the system as a whole; that is, the maximum time between the arrival of a request and the sending of the corresponding response

p_m the amount of memory required for a buffer of incoming requests.

First, it is clear that we can map this task to the problem treated in Theorem 3 by setting $C = R$ and $\tilde{T} = t_e$. Therefore, $n = \lceil t_e/T \rceil$. This means, we need to ask the underlying resource management system (e.g., [8]) to schedule n tasks with a period and relative deadline as well as worst-case execution time of t_e. Of course, this would mean 100% resource utilisation for each CPU, which may not be adequate in a system that should also serve other requests. We can mitigate this by choosing period and relative deadline $t_d \geq t_e$ such that a reasonable CPU utilisation results. We must then set $\tilde{T} = t_d$ and thus $n = \lceil t_d/T \rceil$. Theorem 3 then tells us the required buffer size p in slots à one request. Thus, $p_m = m \cdot p$. This amount of memory needs to be reserved, again, by the underlying resource management system. Finally, $t_r = t_d + t_w$.

We cannot make arbitrary choices for t_d because t_r will increase as t_d increases. Therefore, an upper limit for t_d is given by the response time our container strategy has to guarantee.

5. Related Work

Even though a lot of approaches have been developed for scheduling real-time components (e.g., [4]), only very few approaches attempt to provide automatisms for determining the required resources and component instances. [16] presents an approach for multiplexing TCP-based requests on multiple instances of the same application. Resource allocation and the management of the number of instances is controlled by a closed loop control circuit implemented as an extension of the Linux kernel. Real-Time interfaces [15] are an approach for specifying and reasoning about real-time properties in a componentised manner. Although our formal foundations are related, we aim at automating the runtime support for real-time properties and removing the burden of their handling from the application developer. This is not considered in [15]. The QuA project [12] advocate the notion of *safe composition*, which essentially means that components should be reusable in any context. This is to be achieved by moving all context-related management logic into the infrastructure—for example, resource allocation and instance management. This approach—similar to the concept of container-managed quality assurance presented in this paper—would, thus, form a good framework for the implementation of our ideas. Container-Managed quality assurance has also been discussed to some degree in [9], where we have shown how it is embedded in continuous support for real-time properties from inception to deployment.

The notion of jitter-constrained periodic streams (JCS) has been introduced [5, 6] to subsume several parameter sets describing sequences of events which occur principally with constant rate, but may vary within given limits. Examples for such parameter sets are the traffic description used in ATM connections [14] and the model of linear bounded

arrival processes for transferring continuous media [1]. The JCS model allows to investigate the equivalence of different sets of parameters and to transform descriptions between these parameter sets. Furthermore, the approach is used for buffer dimensioning in an experimental real-time operating system [7] and in data base environments where chains of converters produce and consume sequences of data packets of varying sizes and at varying time instants [2, 11].

Another approach similar to JCS is the so called network calculus [3]. It arose from a set of developments that provide insights into flow problems encountered in the Internet and in intranets and enables the solution of problems of buffer and delay dimensioning. The mathematical foundation lies in the theory of Min-Plus algebras. In contrast to JCS, the model focuses on continuous arrival and consumption processes.

6. Conclusion

We have presented one strategy of container-managed quality assurance, a technology in which component containers manage resources and available software components in such a way as to provide certain services with certain guaranteed nonfunctional properties. Specifically, we have shown how a container can compute the required number of component instances, the memory needed for buffering incoming requests, and the processing-resource demand of an application. All it requires to do so, is information about the worst-case execution time of the component, the response time that should be guaranteed, and the request stream. We have used the mathematical theory of jitter-constrained streams for modelling request streams and deriving buffer size, instance count, and response time.

In this paper, we have developed the theory for this container strategy. We are currently working towards a prototype to test out our theory. Further, we want to extend theory and prototype to cover more advanced scenarios and scheduling strategies.

Acknowledgements

This work has been partially funded by the German Research Council. The authors wish to thank Simone Röttger for her helpful comments on a draft version of this paper.

References

[1] D. J. Anderson. Metascheduling for continuous media. *ACM Transactions on Computer Systems*, 11(3):226–252, Aug. 1993.

[2] H. Berthold, S. Schmidt, W. Lehner, and C.-J. Hamann. Integrated resource management for data stream systems. In *Proc. of the 20th Annual ACM Symposium on Applied Computing (SAC 2005)*, Santa Fe, Mar. 2005.

[3] J.-Y. L. Boudec and P. Thiran. *Network Calculus: A Theory of Deterministic Queueing Systems for the Internet*, volume 2050 of *LNCS*. Springer, 2001.

[4] C. D. Gill, D. L. Levine, and D. C. Schmidt. The design and performance of a real-time CORBA scheduling service. *Real-Time Systems*, 20(2):117–154, Mar. 2001. Kluwer Academic Publishers.

[5] C.-J. Hamann. On the quantitative specification of jitter constrained periodic streams. In *Proc. 5th Int'l Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'97)*. IEEE Computer Society, 1997.

[6] C.-J. Hamann. Schwankungsbeschränkte Ströme. Technischer Bericht TUD-FI05-11, Technische Universität Dresden, Aug. 2005.

[7] C.-J. Hamann and L. Reuther. Pufferdimensionierung für schwankungsbeschränkte Ströme in DROPS. In *GI/ITG Fachtagung MMB*, Trier, Sept. 1999.

[8] H. Härtig, L. Reuther, J. Wolter, M. Borriss, and T. Paul. Cooperating resource managers. In *Workshop on QoS Support for Real-Time Internet Applications in conjunction with 5th IEEE Real-Time Technology and Applications Symposium (RTAS99)*, Vancouver, British Columbia, Canada, June 1999. IEEE.

[9] H. Härtig, S. Zschaler, M. Pohlack, R. Aigner, S. Göbel, C. Pohl, and S. Röttger. Enforceable component-based realtime contracts – supporting realtime properties from software development to execution. *Real-Time Systems*, 2006. To Appear.

[10] Object Management Group. Unified modeling language: Superstructure version 2.0, Aug. 2005. OMG, document number formal/05-07-04.

[11] S. Schmidt, T. Legler, D. Schaller, and W. Lehner. Real-time scheduling for data stream management systems. In *Proc. of the 17th Euromicro Conference on Real-Time Systems*, Palma de Mallorca, July 2005.

[12] R. Staehli and F. Eliassen. QuA: a QoS-aware component architecture. Technical Report Simula 2002-12, Simula Research Laboratory, 2002.

[13] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Component Software Series. Addison-Wesley Publishing Company, second edition, 2002.

[14] The ATM Forum. ATM user network interface specification, version 3.0. Prentice Hall PTR, 1993.

[15] E. Wandeler and L. Thiele. Interface-based design of real-time systems with hierarchical scheduling. In *Proc. 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 243 – 252. IEEE Press, Apr. 2006.

[16] R. Zhang, T. F. Abdelzaher, and J. A. Stankovic. Kernel support for open qos-aware computing. In *Proc. 9th Real-Time and Embedded Technology and Applications Symposium (RTAS'03), Toronto, Canada*, pages 96–105. IEEE Press, May 2003.

[17] S. Zschaler. Formal specification of non-functional properties of component-based software. In J.-M. Bruel, G. Georg, H. Hussmann, I. Ober, C. Pohl, J. Whittle, and S. Zschaler,

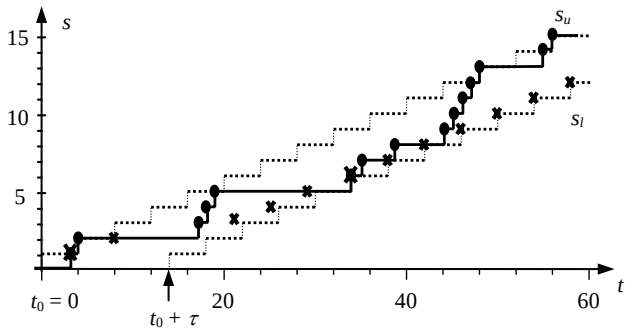

Figure 6. Removal times ✖ for a JCS $S = (4, 1, 14, 0)$ and a sufficiently large buffer

editors, *Workshop on Models for Non-functional Aspects of Component-Based Software (NfC'04) at UML conference 2004*, Sept. 2004. Technical Report TUD-FI04-12 Sept.2004 at Technische Universität Dresden.

[18] S. Zschaler. Towards a semantic framework for non-functional specifications of component-based systems. In R. Steinmetz and A. Mauthe, editors, *Proc. EUROMICRO Conf. 2004*, Rennes, France, Sept. 2004. IEEE Computer Society.

Appendix: Proof of Theorem 1

First, we note that the steps of s_l represent the latest time instants at which a data packet is removed from the buffer. Because of the removal process's periodicity, all previous packets will have been remvoved by this time (cf. Fig. 6). Additionally, no data packet can arrive later than s_l. Furthermore, s_l is identical with the removal process from the time when a data packet arrives at the latest possible time instant. Thus, the required buffer size is essentially the largest vertical distance between s_u and s_l, in other words $p = s_u(t_0 + \tau)$.

More precisely: If $\tau/T \notin \mathbb{N}$ then (considering (3))

$$p = s_u(t_0 + \tau) = 1 + \left\lfloor \frac{\tau}{T} \right\rfloor = \left\lceil \frac{\tau}{T} \right\rceil$$

else if $\tau/T \in \mathbb{N}$ then

$$p = s_u(t_0 + \tau) - 1 = \left\lfloor \frac{\tau}{T} \right\rfloor = \left\lceil \frac{\tau}{T} \right\rceil .$$

It is evident that $t_w = \tau$: s_u represents the earliest possible arrival of a data packet, s_l the latest removal of the same packet. Thus, t_w is the maximum horizontal distance between s_u and s_l.

Figure 6 also shows that these limits are strict: A burst requires p buffer slots and its last data packet is removed from the buffer after a time of τ. ■