Developing a QoS-based Tasklet Trading System

Janick Edinger*, Sebastian VanSyckel*, Christian Krupitzer*, Justin Mazzola Paluska†, Christian Becker*

*University of Mannheim
Schloss, 68161 Mannheim, Germany
{janick.edinger, sebastian.vansyckel, christian.krupitzer, christian.becker}@uni-mannheim.de

†MIT Computer Science and Artificial Intelligence Laboratory
Cambridge, MA, U.S.A.
jmp@mit.edu

***Abstract*—Tasklets provide means for lightweight distributed computing. Based on virtual machines, they allow to utilize excess capacity in cloud computing as well as integrate cloud resources into mobile computing. So far, the Tasklet architecture only provides a best-effort computing abstraction while using direct addressing of peers. However, a higher level of service is often desirable, as is found in other technologies for distributed computing. For example, in the Internet protocol suite, the Transport layer provides the choice of reliable or unreliable communication on top of the best-effort Internet layer. In middleware technology, distribution transparency masks where a specific service is located. Similar guarantees and abstractions in other systems are achieved via Quality of Service (QoS) parameters and mechanisms. In this paper, we discuss QoS categories for Tasklets, develop the necessary system extensions for QoS integration, and present a QoS-based Tasklet trading system. We evaluate our approach in a prototype system by trading Tasklets of two different prioritization levels.**

***Keywords*—Quality of Service (QoS); QoS-based Trading; Tasklets; Cloud Computing**

I. Introduction

Tasklets are inspired by the flexibility of the Internet protocol suite. Based on a lightweight packet exchange protocol (the Internet Protocol), higher layers realize different service qualities, e.g., reliable or unreliable communication using Datagrams or the Transmission Control Protocol, respectively. In a similar way, the Tasklet architecture offers the execution of code chunks in the cloud, thereby utilizing excess capacity as well as integrating cloud resources into mobile computing scenarios. However, especially in mobile settings, the initiator of a Tasklet cannot rely on guarantees regarding the Tasklet's successful execution. The reasons for an unsuccessful execution can differ and depend on the concrete setting. For example, a Tasklet that is executed using excess capacity of a cloud instance may simply not have been completed when this excess capacity has run out. However, the Tasklet's execution may also have finished successfully, but the result may not have been delivered due to a broken link. From the perspective of a client, this will look the same as an unsuccessful execution in the first example. Both situations require different precautions. In the first scenario, binding reservations, i.e., contracts of the computing capacity, or choosing an computing entity with a higher excess capacity than needed may lead to success. In the second scenario, the Tasklet's initiator could either tolerate delays or chose to request a replicated execution in order to ensure the availability of the result. In any case, the Tasklet architecture requires means for expressing and requesting guarantees, in order to reach beyond a best-effort service. Then, a trading service distributes Tasklets to peers that fulfill the requested guarantees. This approach keeps the programming abstraction consistent with the distribution transparency the Tasklet system aims to provide.

In this paper, we present the integration of *Quality of Service* (QoS) into the Tasklet architecture. With these QoS parameters, developers can specify *how* their Tasklets should be executed – e.g., regarding speed, cost, or privacy – instead of relying on the previous best-effort service level. We discuss a set of QoS categories we believe are beneficial for Tasklets and show how to integrate the respective parameters into the Tasklet's code. Further, we introduce a Tasklet trading service that offers location-transparent distribution of Tasklets based on the requested QoS. That is, the trading service chooses an executing peer depending on the QoS specifications in the Tasklet. Finally, we evaluate our approach by examining the distribution and execution of Tasklets in our prototype system that implements the QoS category *priority*.

The remainder of the paper is structured as follows. First, we discuss our system model in Section II, i.e., introduce the Tasklet architecture based on the *chunks* structure. Afterwards, we discuss QoS for Tasklets in Section III. In Section IV, we show how to integrate QoS parameters into the chunk structure, and present our trading service for QoS-based Tasklet distribution. In Section V, we describe implementation details of our prototype and evaluate our approach. We discuss related work in Section VI, before closing in Section VII with a conclusion and outlook on future work.

II. System Model

In this section, we discuss the system model of our work. First, we give a quick overview of the Tasklet architecture and the chunk structure. Then, we show how Tasklets are created and executed, in order to present the integration of QoS parameters in Section IV.

A. Tasklets and Chunks

The *Tasklet* architecture is a fine-grained cloud computing model [1]. A Tasklet itself is a lightweight abstraction for a thread of computation that can be executed on virtual

978-1-4799-2736-4/14/$31.00 ©2014 IEEE

machines. In that, Tasklets realize closures, i.e., they come with everything that is needed to execute them. This includes processing instructions, the data used during execution, and a machine runtime state. A Tasklet is comprised of a graph of linked *chunks*, the underlying representation of code and data. Chunks are ordered, fixed-size arrays consisting of fixed-size, typed slots. Since Tasklets are self-contained computations and include their own memory, they can be migrated from one virtual machine to another without losing their runtime state. They typically incorporate one single task like solving a mathematical task, applying a filter to a picture, or running speech recognition on a single request. In this way, Tasklets allow for fine-grained execution.

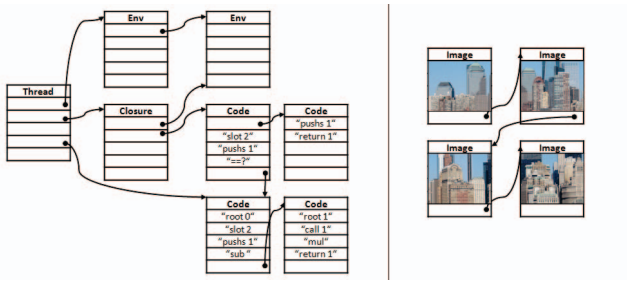


Fig. 1: Examples for data stored in the chunk structure [2].

Figure 1 shows two examples of data stored in chunk structures. The graph on the left hand side shows a representation of compiled programming code ready for execution, whereas the right hand side shows how an image file is split up into parts and stored in a chunk structure.

B. Tasklet Creation and Execution

Tasklets are created dynamically at runtime, depending on which instruction(s) are to be executed on which data. Hereby, the entire process is masked from the user. Upon a user performing an action within an application, the chunk structure is compiled from the respective *JSLite* code – a JavaScript-like language we developed in the chunks project – implementing that action as well as the referenced data. The resulting linked chunk graph, which represents the Tasklet, is stored in a *chunk store*.

From there, chunks can be copied and transferred to other chunk stores using the *chunk exchange protocol* (CXP). Entities that are running a virtual machine to execute Tasklets are called *Tasklet container*, or simply container. A container can run on any suitable hardware that is connected to the Internet. However, one physical machine can also host several containers. A chunk peer contains the chunk store and provides functionality to communicate with other chunk peers in other containers. The tasks of chunk peers include bootstrapping, adding other chunk peers to the overlay, and retrieving and exchanging chunks via CXP.

III. QoS for Tasklets

So far, the Tasklet concept does not provide any guarantees regarding the execution of a Tasklet. Neither does it support non-functional requirements. In this work, we integrate QoS parameters into the Tasklets' code, as well as extend the Tasklet architecture by QoS handling, while maintaining distribution transparency. In the following, we discuss a set of QoS categories we believe beneficial for the use of Tasklets. Please note that the set is not exhaustive, but should be extended with the experience of future work.

Priority. A container can offer its service to various applications simultaneously. Therefore, a container has a waiting queue for managing incoming Tasklets. For background tasks the result of the execution might not be time-critical. For some applications, however, delays are not acceptable, e.g., for any application that involves direct interaction with the user. The boolean QoS *priority* signals the urgency. Containers that support priority Tasklets have an additional queue for prioritized Tasklets that then skip waiting *economy*, i.e., non-prioritized Tasklets. In a commercial system, the priority flag may be coupled with an extra premium for preferred execution.

Speed minimum. The possibility of executing a Tasklet on a faster machine can be very beneficial, especially in mobile computing scenarios with limited resources. However, we can not simply assume that execution in the cloud is faster. In some cases, the advantage of offloading is surpassed by a faster local execution with less costs. Using QoS we can assure the benefit by specifying a *speed minimum*, which guarantees that the chosen offloading container offers a superior performance. The unit can either be absolute (e.g., MIPS) or relative compared to the execution performance of the mobile device.

Cost limit. Tasklet containers can either offer processing capacities for free or may charge for executing Tasklets. In the latter case, users are motivated to reduce the costs of the execution. It is assumed that (i) cost structures of containers are comparable, (ii) they charge users in a pay-as-you-go fashion, and (iii) cost estimates can be done reliably before the execution of a Tasklet. Users can then cap the costs of the execution by setting a *cost limit* as a QoS parameter. The unit of the costs depends on the underlying economic model.

Fault tolerance. Up to now, Tasklets were executed in a best-effort manner, meaning that they might or might not be executed. There are two main reasons, why an execution might fail. First, the executing host crashes or runs out of excess capacity. Second, the application host loses connection to the network and does not receive the result of the Tasklet. For many applications, this unreliability is not sufficient. Hence, *fault tolerance* is a desirable guarantee for the Tasklet system. It can be achieved by running a Tasklet on multiple containers, or, if possible, on the local container as a backup.

Privacy. In order for the Tasklet system to be viable for some applications, the data and computation must be private. That is, confidential data must be sent over the network and both the data and the results must not be leaked. The QoS *privacy* addresses this problem. With it, the transmission can be encrypted by using a secure connection, e.g., Secure Shell (SSH). The application and the container initialize a secure connection via SSH, which is used for communication and transfer of Tasklets. The QoS parameter specifies the SSH protocol version. Further, certificates may be used to ensure the executing container is trustworthy or authorized. This could include Tasklets that contain sensitive data and should only be executed on company devices, which hold a certain certificate. Using the QoS *privacy* alongside with the authorization signature as a parameter, only entitled containers would be selected for the execution of the respective Tasklet.

IV. QoS-based Tasklet Trading

In order to build a QoS-based Tasklet trading system, we need to integrate the QoS parameters into the chunk structure, and design a trading system that distributes the Tasklets depending on these QoS requirements.

A. QoS Integration into Tasklets

Since Tasklets are closed entities, any extension to the architecture must stay within the boundaries of the chunk concept. Thus, to integrate QoS into the Tasklet system, we extended the chunk graph of a Tasklet by *QoS chunks*, which each hold a QoS category and the corresponding parameter(s). For the QoS integration, three steps were necessary: (i) providing an interface to programmers, (ii) extending the JSLite grammar, and (iii) extending the JSLite compiler.

We defined an interface to set the QoS requirements, which – for example for the QoS *priority* – looks as follows:

$$@QOS\ Priority('True'); \quad (1)$$

Listing 1 shows the extended JSLite grammar. The QoS requirements are defined directly by the programmers in the JSLite code. There can be either zero, one or multiple QoS requirements at the beginning of a JSLite program. The token `@QoS` indicates the beginning of the QoS definition and is followed by a comma-separated list of QoS requirements. Each QoS requirement consists of a QoS category and a number of corresponding parameters, which are appended in brackets. The compiler checks whether the QoS categories are valid and notifies the developer in case a QoS category is unknown. The list of possible QoS categories is not final and can be extended when further QoS mechanisms are implemented. At compile-time, each QoS requirement is converted into a QoS chunk with *n* slots. The first slot of each QoS chunk holds the QoS category. Slots 2 to *n*-1 are reserved for the QoS parameters and slot *n* holds the link to the next QoS chunk. If there is no further QoS chunk, slot *n* remains empty.

```
1 program
2     : LT* (qos LT*)? statement (LT* statement)* LT* EOF
3       -> ^(PROGRAM (qos)? statement+)
4     ;
5
6 qos
7     : '@QOS' single_qos (',' single_qos)* terminator
8       -> ^(@QOS single_qos (single_qos)*)
9     ;
10
11 single_qos:
12     : qos_category '(' parameter (',' parameter)* ')'
13       -> ^(qos_category parameter (parameter)*)
14     ;
15
16 qos_category
17     : Priority
18     | SpeedMinimum
19     | CostLimit
20     | FaultTolerance
21     | Privacy
22     ;
23
24 parameter
25     : string_literal
26     | INT
27     ;
```

Listing 1: QoS Specification Syntax in JSLite Grammar.

B. Tasklet Trading System

In general, Tasklets can be executed on any container that offers its resources. However, depending on the QoS requirements, some containers might be more suitable than others for a specific Tasklet. The goal of the trading system is to identify and select appropriate containers for Tasklet execution, as well as to establish a connection between the application and the selected container. Containers can offer their services to more than one application, and applications, in turn, can have more than one registered container. Instead of connecting applications and containers in an $n : m$ fashion, we introduce traders, which act as an intermediate brokers between the two entities. This way, we reduce the amount of overhead for applications since they do not have to deal with container registration or selection mechanisms anymore. Furthermore, containers do not need to know the address of every single application but only connect to one or more traders.

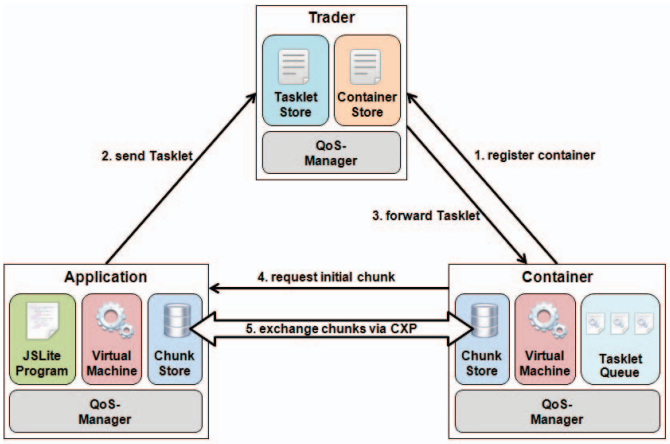


Fig. 2: Overview of the Tasklet trading system.

Figure 2 shows the operating principle of the trading system at a high level view. The trader itself has neither a chunk store nor a virtual machine. This is due to the fact that it does not execute Tasklets itself and, thus, does not have to be part of the chunk peer overlay.

First, containers register their resources at a trader, which maintains a list of available containers – the *container store*. During registration, the container transmits all its QoS capabilities like processing speed, pricing and available encryption methods to the trader. Second, the application creates a Tasklet which might include QoS requirements. It sends the Tasklet to the trader where it is added to a list of pending Tasklets – the *Tasklet store*. Third, the trader selects the most suitable container for the Tasklet from the container list, using the containers' QoS capabilities. Instead of sending information about the container back to the application, the trader forwards the Tasklet directly to the container. This reduces the communication overhead for the application. Containers can accept more than one Tasklet at a time. The Tasklets are stored in a queue and are executed in a FIFO order. When the container dequeues a Tasklet, it contacts the application in the fourth step and requests the chunks that are necessary to execute the Tasklet. During Tasklet execution, the application and the container keep exchanging the required chunks. If the trader

can not find any suitable container for the execution, it has two options. It either holds the Tasklet in a storage and periodically checks for containers until a timeout occurs, or it immediately returns the Tasklet back to the application, where it either might be executed locally or is sent to a trader another time.

Instead of sending the whole Tasklet to the trader, the applications only transmits the *signature* of the Tasklet. The signature encodes all necessary information of a Tasklet, i.e., its ID, the host and port of the application, and all QoS parameters. This avoids additional data transfer for the application and reduces complexity at the trader side. Further on, the trader forwards the signature of the Tasklet to a container, which can use the Tasklet ID to bootstrap the chunk graph of the Tasklet. Thus, only the application and the container exchange chunks via CXP.

All three entities contain a QoS manager. In the application it is implemented by the JSLite compiler, which creates the QoS chunks. The QoS manager in the trader is in charge of selecting the most suitable container depending on the QoS requirements of the Tasklet, e.g., selecting the cheapest container that exceeds the speed minimum and is within the specified cost limit. At the container side, the QoS manager enforces the QoS requirements of the Tasklet, e.g., gives them priority over other Tasklets or initiates a secure connection for data transfer to the application.

V. Implementation and Evaluation

There are multiple QoS categories that can be implemented into the Tasklet system. We discussed some QoS in Section III. Further QoS are possible as well. Without any QoS specification, Tasklets are executed in a best-effort fashion. Currently, it is hard to estimate the number of required chunks prior to the execution of a Tasklet. Thus, the amount of data that needs to be sent over the network is not known a priori. The same restriction holds true for the computational complexity of a Tasklet. Hence, reliable estimates needed for some QoS are not yet feasible, which is why we chose to evaluate the system using the QoS *priority* as an example. In the following, we give some implementation details of our trading system, as well as evaluate the process of QoS-based Tasklet trading.

A. Implementation

The trading system was implemented in Python 2.7. The chunk system makes use of Autobahn Websockets 0.6.4 for Python [3] and Twisted 13.1.0 for Python [4]. Each container and each application run in their own processes which are spawn by a container and application factory.

1) Application: The applications create Tasklets from JSLite code and send the signature of each Tasklet as a string to a trader. In our simulation, the Tasklets were similar to each other to maintain comparability. Application nodes could serve as containers as well and execute Tasklets from other applications or even their own ones.

2) Trader: The trader distributes the Tasklets depending on their respective QoS parameters, as well as on the capabilities of the registered containers. That is, the trader examines the QoS parameters in the order given by the developer and matches them with the capabilities of the containers. Hence, with each QoS the list of suitable containers gets more and more restrictive. In our system, lookup of a suitable container is almost instant. However, our prototype does not allow a container to change its QoS capabilities without re-registering with the trader. As all Tasklets in our evaluation are equal, no further load balancing is required.

3) Container: At each container, Tasklets are executed in FIFO order. Further, only one Tasklet is executed at a time. Hence, the waiting period for a Tasklet is determined by the workload of the container it has been assigned to. Introducing a simple model of execution levels, we distinguish among two different queues. In order to execute the incoming Tasklets based on their priority declaration, the modifications of the containers is minimal. Instead of holding only one queue for Tasklets, each container holds two queues – the *economy queue* and the *priority queue*. After completing the execution of a Tasklet, the container first checks the priority queue for further assignments. Only if the priority queue is empty, the container checks and processes its economy queue. Hence, as long as a container has prioritized Tasklets in its queue, the economy Tasklets are neglected. In case both queues are empty, the container will proceed with the next incoming Tasklet, whether priority or not. As there are no guarantees for the best-effort Tasklets, priority execution also does not guarantee an immediate execution or a maximum waiting period, but ensures that each priority Tasklet is processed before the next economy Tasklet is executed.

B. Evaluation

In our evaluation, we distributed the different components of the system to two PCs as well as one blade server connected via our department Ethernet. The applications were executed on a Lenovo T430s laptop equipped with a 64-Bit Windows 7 Professional, an Intel i7-3520 Dual-Core 2.9 GHz CPU, and 8 GB RAM. For the containers we used a desktop PC running a 64-Bit Windows 8 OS equipped with an Intel i5-2500K Quad-Core 3.3 GHz CPU and 8 GB RAM. Finally, the single trader ran on the blade server with a 2x Quad-Core Intel Xeon with 2.33 GHz CPU, 6 GB RAM, and a 64-Bit Windows Server 2008 Standard Edition. During the evaluation, the applications repeatedly issue Tasklet execution requests to the trader based on the parameters of the specific run, which determined the number of Tasklets per application, as well as the lower and upper bound of a random sleep period between the requests.

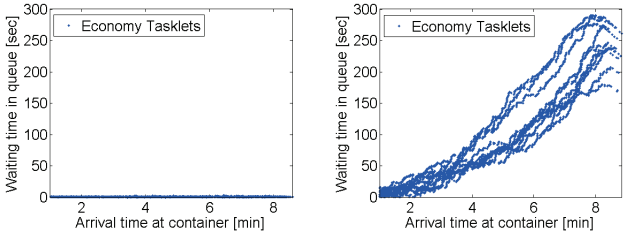
TABLE I: Evaluation scenarios

Scenario	1a	1b	2a	2b	3
# of applications	20	20	20	20	10 + 10 after 120s
# of Tasklets per application	100	200	100	200	200 / 200
# of containers	10	10	10	10	10
Seconds between requests	2-8	1-4	2-8	1-4	3-9 / 1-3
Economy/priority ratio	100/0	100/0	80/20	80/20	80/20

We evaluated the process of Tasklet trading and execution in five different scenarios. Table I summarizes their settings. The first two scenarios (1a, 1b) serve as a benchmark and only involve economy Tasklets, meaning that each Tasklet has the same priority. Containers strictly stick to the FIFO order

for Tasklet execution and there is no way to avoid a delayed execution of a Tasklet in case of a congested system.

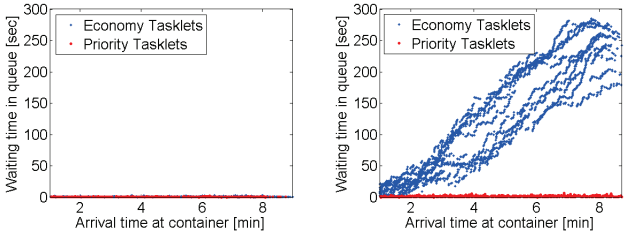
In the second two scenarios (2a, 2b), we use the same basic settings as before, but label 20% of the Tasklets as priority Tasklets. In both, Scenario 1a and 2a, we underutilize the capacity of the containers – i.e., the containers can execute the Tasklets faster as they come in – in order to examine the difference in waiting time between economy and priority Tasklets. In scenarios 1b and 2b, on the other hand, we increase the number of Tasklets per application, as well as reduce the waiting period between Tasklet execution requests, in order to overutilize the containers. Here, the waiting time of the economy Tasklets should increase heavily. Scenario 3 features both under- and overutilization in order to see how the system recovers from overload. That is, we create a steady workload that underutilizes the containers comparable to Scenario 2a. After 2 minutes, we start a second set of applications and flood the containers with Tasklet execution requests, in order to create a peak in the workload. When the second set of applications is done sending Tasklets, we expect the containers to slowly work off the build-up. During the surge, we still expect priority Tasklets to be executed with a minimal waiting period.



(a) Underutilization (1a). (b) Overutilization (1b).

Fig. 3: Waiting period per Tasklet in Scenario 1a and 1b.

Figure 3 shows the waiting period of the Tasklets in Scenarios 1a and 1b during underutilization (Figure 3a) and overutilization (Figure 3b). Whereas in the first case, Tasklet execution is almost immediate, the waiting period increases linearly when the queue for economy Tasklets builds up in size during overutilization.



(a) Underutilization (2a). (b) Overutilization (2b).

Fig. 4: Waiting period per Tasklet in Scenario 2a and 2b.

Similar to above, Figure 4 shows the waiting periods of the Tasklets in Scenario 2a and 2b. During underutilization (Figure 4a) Tasklet execution is almost immediate, and there is no great difference in the waiting periods of the economy and priority Tasklets. During overutilization (Figure 4b) we see that even though the containers can not keep pace with their economy queue, the waiting time for priority Tasklets stays consistent small as intended. Naturally, this would no longer be the case if the amount of priority Tasklets alone exceeds the container capacities.

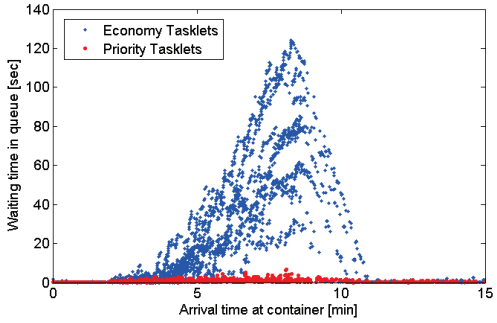


Fig. 5: Waiting period per Tasklet in Scenario 3.

Figure 5 shows the waiting period per Tasklet in Scenario 3. At first, the containers are not fully utilized and so the waiting periods for both economy and priority Tasklets are very short. After two minutes, the second set of applications is launched and the number of Tasklet execution requests increases enough to exceed the computing capacity of the containers. However, as in Scenario 2b, the priority Tasklets are still executed almost instantly, whereas the economy Tasklets' waiting periods increase dramatically and peak at about the eight minute mark, at which the second set of applications is done requesting Tasklet executions (see Figure 6). Following this peak, the system recovers and work off the build-up.

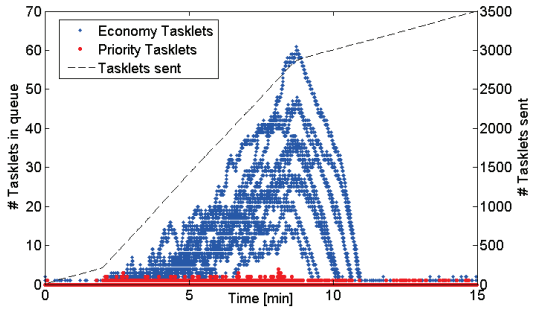


Fig. 6: Queue sizes and accumulative requests in Scenario 3.

Figure 6 shows the respective queue sizes with regard to the accumulative number of requests (dashed line) in Scenario 3. The behavior of the queue sizes is consistent with the waiting periods shown in Figure 5. The size of the economy queue increases dramatically with the start of the second set of applications, while the size of the priority queue stays almost constant.

Overall, our evaluation shows that priority Tasklets are always executed after a consistently short delay, even though the overall workload exceeds the capacity of the containers. Furthermore, the results show correctness of our overall approach to QoS-based Tasklet trading.

VI. Related Work

There are several approaches that are concerned with distributed execution of programming code ([5], [6], [7]). Similar to the Tasklet architecture these systems facilitate the remote execution of code transparently. However, they merely deal with non-functional requirements as QoS and do not consider any mechanisms to select suitable execution hosts.

Nahrstedt et al. provide a detailed overview of approaches to integrate QoS into distributed middleware [8]. There are different aspect-oriented approaches to enrich CORBA by a QoS management. We extended the OMG IDL to a QoS-aware interface definition language (QIDL) [9] and used it in a Management Architecture for Quality of Service (MAQS) [10]. Loyall et al. presented the Quality Object framework [11]. Developers can use quality description languages to state their QoS requirements. The QoS-aware objects are then handled and places by delegates. Hauck et al. provides a generic interface to integrate and control QoS in an object middleware on top of CORBA [12] . For each QoS category one client stub is created which enforces the guarantees. The required stub is chosen at runtime.

Nahrstedt and Smith developed the concept of a *QOS Broker* to provide applications with end-to-end guarantees and negotiation mechanisms for QoS [13]. The principle of an intermediary instance based on QoS properties can be found for web service applications, too. Ran proposes to use an intermediary for choosing web services based on functional as well as QoS properties [14]. A similar approach is presented by Tian et al. [15]. Other authors propose solutions with additional functionality for dynamic QoS negotiation between service providers and consumers and QoS monitoring [16] or QoS-aware resource allocation of service provider's resources [17]. Zeng et al. developed a QoS-aware middleware which maximizes the overall quality of web service compositions [18]. There are several implementations of trading services referring to the Reference Model of Open Distributed Processing (RM-ODP) [19]. A prominent example of a trader-based architecture is Jini [20].

VII. Conclusion and Outlook

In this paper, we presented a QoS-based Tasklet trading system. First, we discussed possible QoS categories that are beneficial in certain situations. Afterwards, we showed how to integrate QoS parameters into the chunk system, and presented our QoS-based Tasklet trading service. With our approach, developers can specify a set of QoS parameters that are important to the Tasklets an application needs to execute. The trading service then matches the requested set of QoS parameters with the capabilities of Tasklet containers in the cloud. Finally, we evaluated our approach using the QoS category priority as an example.

In future work, we will improve our prototype by implementing further QoS categories, as well as introduce load balancing mechanisms into the trader. Further, we want to address mobile computing scenarios. We want to integrate offloading decision mechanisms based on parameters such as network bandwidth, link type, and battery level, as well as address aspects of discontinued operations. Finally, containers should be monitored regarding the QoS capabilities they claim to provide. Hence, we plan on exploring mechanisms for assessing the trustworthiness of containers, e.g., peer-to-peer based solutions.

Acknowledgment

This work was supported by the German Academic Exchange Service (DAAD).

References

[1] J. Mazzola Paluska, H. Pham, G. Schiele, C. Becker, and S. Ward, "Vision: a lightweight computing model for fine-grained cloud computing," in *Proc. of the 3rd ACM workshop on Mobile cloud computing and services*. ACM, 2012, pp. 3–8.

[2] J. Mazzola Paluska, H. Pham, and S. Ward, "Chunkstream: Interactive streaming of structured data," *Pervasive and Mobile Computing*, vol. 6, no. 6, pp. 607 – 622, 2010.

[3] Tavendo, "Autobahn websockets," 2013, accessed Nov. 10, 2013. [Online]. Available: http://autobahn.ws/

[4] TwistedMatrixLaboratories, "Twisted," 2013, accessed Nov. 10, 2013. [Online]. Available: http://twistedmatrix.com/

[5] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "Maui: making smartphones last longer with code offload," in *Proc. of the 8th international conference on Mobile systems, applications, and services*. ACM, 2010, pp. 49–62.

[6] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "Clonecloud: elastic execution between mobile device and cloud," in *Proc. of the 6th conference on Computer systems*. ACM, 2011, pp. 301–314.

[7] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang, "Unleashing the power of mobile cloud computing using thinkair," *CoRR*, vol. abs/1105.3232, 2011.

[8] K. Nahrstedt, D. Xu, D. Wichadakul, and B. Li, "Qos-aware middleware for ubiquitous and heterogeneous environments," *Communications Magazine, IEEE*, vol. 39, no. 11, pp. 140–148, 2001.

[9] C. Becker and K. Geihs, "Quality of service-aspects of distributed programs," in *Int. Workshop on Aspect-Oriented Programming*, 1998.

[10] C. Becker and K. Geihs, "Generic qos-support for corba," in *Proc. of the 5th Int. Symposium on Computers and Communications*. IEEE, 2000, pp. 60–65.

[11] J. Loyall, D. Bakken, R. Schantz, J. Zinky, D. Karr, R. Vanegas, and K. Anderson, "Qos aspect languages and their runtime integration," in *Languages, Compilers, and Run-Time Systems for Scalable Computers*, ser. LNCS. Springer, 1998, vol. 1511, pp. 303–318.

[12] F. J. Hauck, U. Becker, M. Geier, E. Meier, U. Rastofer, and M. Steckermeier, "Aspectix: A quality-aware, object-based middleware architecture," in *Proc. of New Developments in Distributed Applications and Interoperable Systems*. Springer, 2001, pp. 115–120.

[13] K. Nahrstedt and J. M. Smith, "The QOS Broker," *IEEE MultiMedia*, vol. 2, no. 1, pp. 53–67, 1995.

[14] S. Ran, "A Model for Web Services Discovery With QoS," *ACM SIGecom Exchanges*, vol. 4, no. 1, pp. 1–10, 2003.

[15] M. Tian, A. Gramm, T. Naumowicz, H. Ritter, and J. Schiller, "A concept for qos integration in web services," in *Proc. of the 4th international conference on Web information systems engineering workshops*. IEEE, 2003, pp. 149–155.

[16] M. Serhani, R. Dssouli, A. Hafid, and H. Sahraoui, "A QoS broker based architecture for efficient Web services selection," in *Proc. of the Int. Conference on Web Services*. IEEE, 2005, pp. 113–120.

[17] T. Yu and K.-J. Lin, "The Design of QoS Broker Algorithms for QoS-Capable Web Services," in *Proc. of the Int. Conference on e-Technology, e-Commerce and e-Service*. IEEE, 2004, pp. 17–24.

[18] L. Zeng, B. Benatallah, A. Ngu, M. Dumas, J. Kalagnanam, and H. Chang, "Qos-aware middleware for web services composition," *IEEE Transactions on Software Engineering*, vol. 30, no. 5, pp. 311–327, 2004.

[19] K. Raymond, "Reference model of open distributed processing (rm-odp): Introduction," in *Open Distributed Processing*. Springer, 1995, pp. 3–14.

[20] J. Waldo, "The jini architecture for network-centric computing," *Communications of the ACM*, vol. 42, no. 7, pp. 76–82, 1999.