Proceedings of the 18th Conference on Computer Science and Intelligence Systems pp. 699–709

DOI: 10.15439/2023F8957
ISSN 2300-5963 ACSIS, Vol. 35

Multi-queue service for task scheduling based on data availability

Kamil Rybiński, Michał Śmiałek
0000-0002-5543-790X, 0000-0001-6170-443X
Warsaw University of Technology
pl. Politechniki 1, 00-661 Warszawa, Poland
Email: {michal.smialek, kamil.rybinski}@pw.edu.pl

***Abstract*—Large-scale computation (LSC) systems are often performed in distributed environments where message passing is the key to orchestrating computations. In this paper, we present a new message queue concept developed within the context of an LSC system (BalticLSC). The concept consists in proposing a multi-queue, where queues are grouped into families. A queue family can be used to distribute messages of the same kind to multiple computation modules distributed between various nodes. Such message families can be synchronised to implement a mechanism for initiating computation jobs based on multiple data inputs. Moreover, the proposed multi-queue has built-in mechanisms for controlling message sequences in applications where complex data set splitting is necessary. The presented multi-queue concept was successfully implemented and applied in a working LSC system.**

I. Introduction

LARGE-SCALE computations (LSC) are often performed in a distributed environment. Several computation nodes can be linked together to execute resource-consuming tasks. One of the main issues is the management of data flow between these nodes [1]. The goal is to reduce data transfer overheads and maximise the speed of computations. In a data flow-driven approach to LSC, computation applications are divided into computation steps, with data flowing as messages between these steps. An example of such a system is the BalticLSC platform [18], [13] (www.balticlsc.eu). It is a low-code computation environment created as part of a project intended to facilitate easier access to LSC. It performs computations in the form of Docker-based computation modules that are orchestrated according to applications defined in a dedicated graphical language called CAL (Computation Application Language) [19]. CAL revolves around the flow of data. The specific sequence of execution for computation modules is defined by specifying paths through which data flows between them. This creates the need for means to schedule starting of computation jobs according to data availability and to propagate that data between modules as defined by a CAL application. Other examples of systems where computations are driven by flowing data and that use graphical languages are WS-PGRADE [7], [6] and Flowbster [8]. Also, other Scientific Workflow Systems use this kind of approach [11].

This work was partially funded by the InterReg BSR Programme, grant no. #075

Figure 1 shows an example application written in CAL. This application consists of three computation modules (the boxes) communicating through 6 data flows (the arrows). The application has two inputs (“Videos” and “Subtitles”) and one output (“Films”). In this example, the inputs and the output are sequences of folders containing appropriate files. Module “A” is a File Synchroniser type. It accepts two folders and, using certain naming conventions, produces two synchronous sequences of files (here: a video file matched with a subtitle file). The video file is processed by module “B” which is a Video Converter. Finally, module “C” (a VS Mixer) mixes the processed video file with the subtitle file received directly from module “A”. Individual files resulting from module “C” are then placed in appropriate folders at the output.

The flow of data between computation modules in Figure 1 is shown through additional tokens besides data flows. The numbers in circles relate to the specific flows (numbered 1 to 6 to denote the six flows). The numbers in rectangles denote sequence numbers. As we can notice, these sequence numbers can be stacked. For instance, token “1” with sequence number “0” (T1-S0) denotes a token message representing the first folder with video files on the “Videos” input. In case more folders are placed on the input, additional token messages (T1-S1, T1-S2, ...) are produced (not shown in the figure). Token number “3” necessitates a stack of two sequence numbers (e.g. T3-S0-S0, T3-S0-S1, ...). The top element corresponds to the source folder (cf. T1-S0), and the bottom element represents the number in the sequence of files in that folder. Note that these sequence numbers have to be maintained throughout computations. In our example, each instance of module “C” has to receive tokens with appropriate sequence stacks. For instance, token message T5-S0-S1 has to be matched with token message T4-S0-S1.

As in the example above, the message-passing system operates on sequences of token messages. It is thus natural to implement it using queues. However, according to our best knowledge, no existing solution could offer out-of-the-box all the functionality required to schedule jobs controlled by multiple data passing between the jobs. General-purpose queue systems are heavy-weight and offer extensive functionality for instantiating multiple queues. However, they do not provide mechanisms for task scheduling based on many synchronised queues. After analysing the available options, we have decided

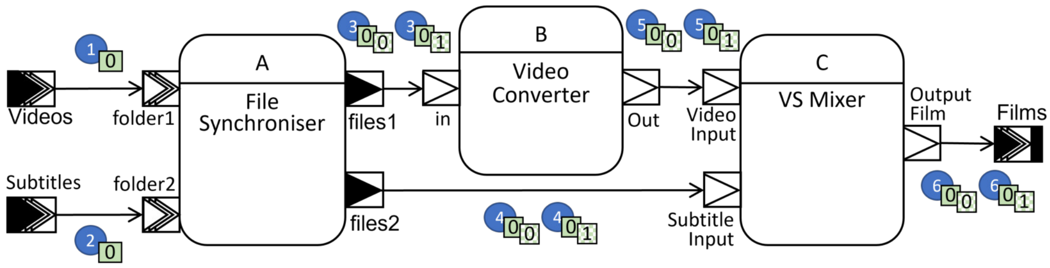


Fig. 1. Example application in CAL with sample tokens

to develop our own lightweight queue system, suitable for the CAL execution environment. In this paper, we present the details of our queue system and its application as part of a computation orchestration system within the BalticLSC environment.

II. Related work

There are two main approaches to performing distributed computations – orchestration and choreography [14], [16]. This follows two global trends of deploying workflows or composite micro-services. In the first approach, an external entity manages the flow of computations. BalticLSC and WS-PGRADE are examples here. In the second approach, this flow is realized by each participating computation element that fulfils its role that is known in advance. Flowbster is an example here.

The usage of FIFO queues is especially important for choreographed computations where they are intensively used. This includes managing the initiation of jobs in the proper order (either directly [9] or as part of more complex mechanisms [12]) and ensuring the exchange of messages between them. Especially relevant in the context of this paper is the usage of systems consisting of multiple queues (multi-queues) for task scheduling [2], [10], [5], [20], [22], [15]. Since such systems use queues to schedule tasks, they operate by scheduling computation steps, not data flows. For example, Li et al. [10] propose a system for scheduling read-write tasks for a distributed database system. Their solution is based on a multi-queue feature built into the Cassandra database system.

Generally, using queues provides two main advantages. Firstly, the responsibility to deliver messages falls on the queues, without the need to implement any additional mechanisms on the side of the message senders. Secondly, queues create separation in time between sending and receiving messages. Senders can send messages to yet non-existent recipients. Moreover, they can finish working before the recipients even start.

When describing a queue, we distinguish two roles of entities interacting with it - producers supplying messages to the queue and recipients interested in receiving these messages. Most queues utilize the publish-subscribe pattern. In this pattern, recipients declare to a queue their interest in messages of a given type. When a message fulfilling appropriate criteria arrives from a producer, all interested recipients are notified. The delivery of messages to recipients is generally done in one of two ways. Either a queue sends messages to its recipients (the push method), or recipients ask a queue for the messages and fetch them themselves (the pull method).

There are also a few additional issues that queues need to address, especially in the case of the push method. Messages can get lost because communication between queues and their recipients is never flawless. This necessitates repeating messages to guarantee their delivery. On the other hand, this can cause occurrences of unwanted duplicated messages. As a result, striving to ensure guaranteed delivery or lack of duplicates leads to hindering one of these features. Repeating undelivered messages could also change the order of messages, that is – the order in which messages reach the recipients can be different than the order in which they arrive from the producers. It is especially problematic when messages are delivered to multiple recipients simultaneously.

There are many systems implementing message distribution using queues [23], [21], but two of them found very wide usage – RabbitMQ [17] and Apache Kafka [3]. RabitMQ is an implementation of an extension to the AMQP protocol [24]. It supports delivering messages through both the push and the pull methods. However, the former is not recommended in most cases. Message distribution in RabbitMQ is oriented around the concept of exchanges. Exchanges define how messages are distributed between queues (which are related to particular recipients). RabbitMQ supports several types of exchanges, which allow directing messages to particular queues (direct exchange), duplicating messages to groups of queues (fanout exchange), distributing messages according to predefined topics (topic exchange) or checking for more complex patterns in message headers (header exchanges). The most commonly used publish-subscribe pattern is typically realized using topic exchanges through recipients subscribing to particular topics. To provide messages to recipients using the push method reliably, RabitMQ introduces a special mechanism for the acknowledgement of received messages by the recipients. It works based on the same principles as the mechanism for acknowledging receiving messages from the producers by the queue system. The idea was introduced to avoid confirming message delivery through the transaction system. Even if more reliable, this was considered as inconvenient for more trivial cases. Its existence also allows more subtle options used

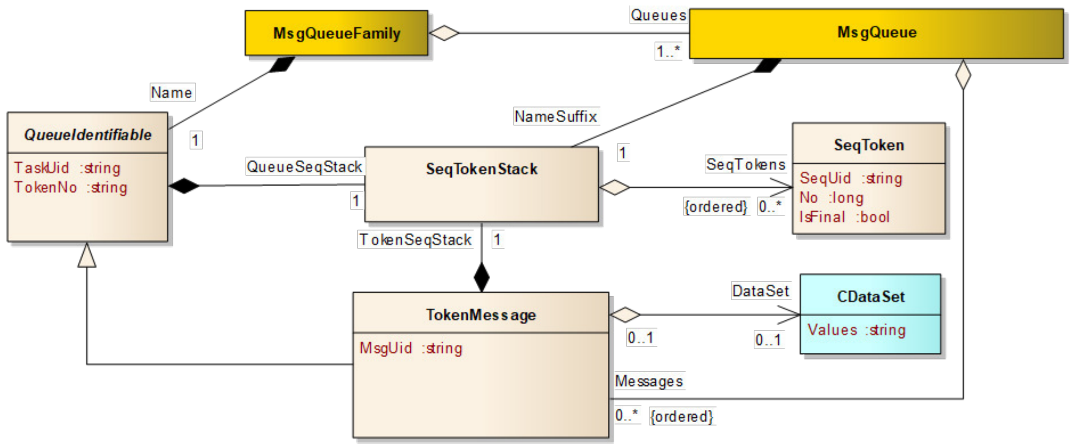


Fig. 2. Multiqueue metamodel

in managing the distribution of messages, like defining the prefetch count – the maximum number of unacknowledged messages that can be sent to a given recipient.

In the case of Apache Kafka [25], [4], queues are organized by topics to which particular recipients can subscribe. Each topic can also be further divided into partitions containing messages for a given recipient. Message delivery is then done by the pull method. This means that particular subscribers need to request messages that are of interest to them, typically as a part of a so-called pull loop. Kafka by default does not delete messages after they are delivered, and there is a possibility to access them again when needed. Accessing messages is done by means of an offset – the particular position in the sequence of messages that indicates which of them were already received. This particular setup obviously makes any message acknowledgement from the recipient obsolete. Thus, Kafka supports only the acknowledgement of message delivery to producers. However, particular client implementations can simulate an acknowledgement mechanism to manage the offset.

III. Multi-queue concept

As mentioned in the introduction, the orchestration of jobs in systems like BalticLSC is based on data flowing between these jobs. This makes message passing extremely important and imposes additional requirements. First, we need to ensure a reliable exchange of messages, as in every distributed system. In addition, our solution needs to allow representing and checking orchestration information attached to the messages. This information can be used to direct messages to appropriate queues.

Figure 2 shows the data structure of our multi-queue in the form of a UML model (a meta-model). The exchange of data between computation modules is handled by "token messages" (see the "TokenMessage" class), and such messages are managed by our multi-queue. Each token message contains a reference that points to a specific data set (see the "CDataSet" class). Data sets can contain data directly or can hold information about accessing data held in a storage system. Each token message is assigned properties that identify its position in the data flow within an instance of a specific application. This is reflected by two attributes specialised by the "TokenMessage" class from the "QueueIdentifiable" class. The "TaskUid" attribute refers to a specific task or – in other words – an instance of an application being executed within the BalticLSC system. The "TokenNo" attribute refers to the particular token number assigned to a specific data flow within the respective application (e.g. tokens 1-6 in Figure 1).

Token messages contain additional information about their sequence numbers. In fact, each token message contains a stack ("SeqTokenStack) of sequence numbers ("SeqToken"). This stack is divided into the "QueueSeqStack" and the "TokenSeqStack". The first stack corresponds to the specific queue to which this message is designated, while the second stack corresponds to a specific sequence of tokens produced by computation modules. These stacks ensure proper processing in applications where data can change its "granularity" (e.g. from folders to files). In such cases, we need to split larger data sets into many smaller pieces to be processed individually (e.g. in parallel). On the other hand, we might need to merge many smaller pieces into a final, larger data set. At the same time, we need to manage these various sequences so that relations between and within these sequences are preserved.

The token message stacks ("SeqTokenStack") allow keeping track of dependencies between data items and thus enable proper distribution of computations between computation jobs. The general rule is that a new level of the stack is added when a data set (file, folder) is split into smaller elements. On the other hand, when several data items are merged, one level of the stack is removed. Note that in some cases, two or more stack levels can be added or removed, depending on particular processing (e.g. processing of a two-dimensional data matrix). Moreover, it can be noted that sequence tokens contain sequence numbers ("No" in the "SeqToken" class). This is necessary due to the possible parallel processing of token messages. In case of delays in the processing of tokens, their order has to be maintained regardless of individual processing times. Moreover, we need a flag denoting the last token in a sequence ("IsFinal").

The queue system that handles token messages consists of two elements. The top-level element is the Message Queue Family ("MsgQueueFamily"), which contains individual Message Queues ("MsgQueue"). The role of the queue family is to handle messages associated with a single data flow (see Figure 1 again). The individual queues handle messages directed to specific computation module instances (jobs) deployed in different distributed computation nodes. Queues are identified by token sequence stacks similar to the token messages that they contain. The identification ("Name") of a message queue family contains two main values describing a token (TaskUid, TokenNo). This is appended with a token sequences stack ("QueueSeqStack") to form the full identifier of a queue family. The message queues contain the suffix part ("NameSuffix") of the token sequence stack, identifying the particular jobs to which tokens should be sent. As we can notice, the main assumption is that individual queues contain token messages that are meant to be processed by the same computation modules. Thus, these token messages point to the same type of data. The assignment of messages to queues is done in two steps. First, we assign them to a message queue family and then – to a particular queue.

Token messages in each of the queues contain full token sequence stacks. In addition to the stack defined in the queue family and its queues, the token sequence stack in a message contains additional levels defining the sequence of tokens sent to the particular job. As a result, before putting a token message to a specific queue, we need to construct a token sequence stack consisting of two parts. The first part identifies the queue (and its family), and the other part contains indexing data that will be forwarded to the recipient (job) to help in internal processing.

In practice, the relationships indicated by the elements of each "SeqTokenStack" serve two purposes. Firstly they ensure that jobs that should process corresponding groups of data from different queues will receive correct data items. Secondly, when computations are distributed between separate machines, related data will be grouped together to reduce unnecessary data transfer. It is also worth mentioning that there is an exception to these rules. This is for so-called "simple" modules, i.e. those that process only one token message from a single queue at a time. In the case of such queues, there is no need to use SeqTokenStack elements for either of the two mentioned purposes, at least at the individual level. However, to maintain consistency in handling queues, we still use "SeqTokenStack" to identify such queues.

An example of a queue family is shown in Figure 3. The outer box represents a family with a task identifier ("Task1"), a token number ("8") and a token sequence stack (here with one level – "0"). The family contains three queues. Each queue contains a "suffix" for the token sequence stack ("0", "1" or "2"). In each queue, we have several token messages. As we can see, each message contains the same token number ("8")

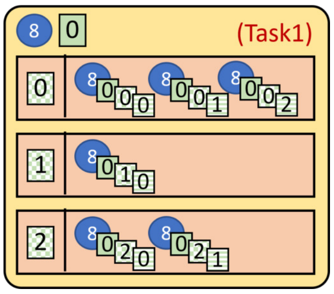


Fig. 3. Example structure of a queue family

and token sequence stack prefix ("0"), which reflects the queue family. Moreover, tokens assigned to each queue in the family have the second levels of their stacks corresponding with their appropriate queue stack suffix. Finally, the third level in the stack reflects the token sequences.

Queues defined according to the presented rules can be used to initiate jobs (instances of computation modules). In general, specific modules require either one or many tokens to be delivered to them from a given queue. For some queues, the arrival of tokens is required to start a job. In other cases, tokens arriving at a queue are passed to a job but are not mandatory to start processing. With our multi-queue, we can easily determine meeting conditions to start jobs. When a new token that matches a given queue in a family arrives, we can check all the queues in queue families assigned to the inputs of a respective computation module. If all the corresponding queues contain tokens, the condition for a new processing job is met. We can then either start a new instance of a computation module or use an instance that is currently idle.

Apart from this additional functionality, our queues work like traditional FIFO queues. They use the publish-subscribe pattern to deliver tokens to jobs using the push method. An additional non-standard element is the implementation of the acknowledgement mechanism, which has some similarities to that of RabitMQ. When a message is delivered to the queue, it is first stored in it. It then waits for its turn (FIFO) and gets assigned to one of the available recipients. This is done by balancing the number of messages each registered recipient receives. Recipients may reject messages (e.g. have insufficient resources), which results in attempting to send the message to another recipient. If the message is accepted, its status changes accordingly, but it is not removed until the recipient acknowledges the finishing of its processing. There is also a possibility of the recipient sending a negative acknowledgement (processing did not succeed). In this case, the message status is reverted, which causes a repetition of the above process.

IV. Illustrative Example

In this example, we use the application presented in the introduction (Figure 1). Each data flow (numbered from 1 to 6) is associated with one or more queue families. Moreover, these families are grouped by computation modules depending on

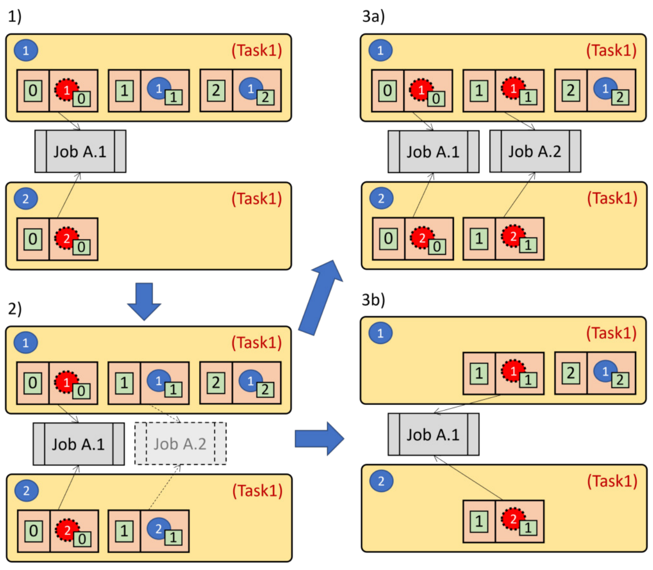


Fig. 4. Queues for tokens 1 and 2

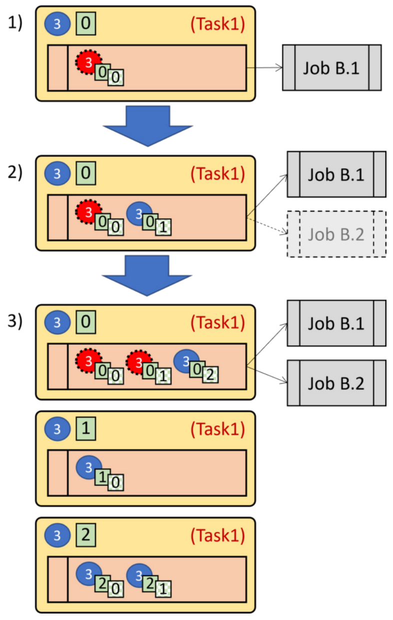


Fig. 5. Queues for token 3

their inputs. These groups can be used to initiate appropriate jobs being instances of the computation modules. We assume that a single task was created from our example application ("Task1"). Obviously, more such tasks can be executed at the same time, and thus more queue families can be created whose identification differs only by the task identifier.

Figure 4 shows queues responsible for providing token messages to instances of module A in our application. The situation shown in step 1) is somewhat advanced in time. We have two queue families (related to tokens "1" and "2"), which already have a few token messages in each of them. We also have one instance of the module (job) running (named "A.1"). It can be noted that each input of module A has been assigned a separate queue family. Moreover, separate queues related to the existing token sequence stacks were created inside the families. At this moment, we have three queues for tokens of type "1" and one queue for tokens of type "2". Note that each of these queues was created with the arrival of an appropriate token message. The name suffixes correspond to the token sequence numbers of these token messages.

When queues 1-0 and 2-0 were created, job A.1 was initiated and assigned to these queues as their recipient. Through this mechanism, the job is guaranteed to get the related pair of token messages. In the situation shown in step 1), we already have job A.1 running and processing the pair of token messages denoted with the sequence number "0" in their token sequence stacks. The queues marked these messages as delivered but did not delete them, as the job did not acknowledge finishing their processing. We have also two additional messages in the first queue family that haven't been delivered yet.

In step 2), a new message is inserted into the second queue family which results in creating an appropriate queue (2-1). With this arrival, we have a new pair of matching token messages (1-1 and 2-1). This creates a condition to create a new job. Depending on the decision of the job broker module, this can lead to one of two situations shown as 3a) and 3b) in Figure 4. In the first situation, a new instance of the computation module A was started ("A.2"). This instance has immediately subscribed to the two related queues. The appropriate messages were then delivered and marked as being processed. In the second situation, a new instance of module A was not created. The job broker waited until instance A.1 finished its previous processing. Only then could the instance subscribe to queues 1-1 and 2-1. This is followed by delivering the new pair of token messages to A.1 and marking them as being processed. Note that the second situation can occur in the situation of limited resources. If we cannot create a new instance of module A, we must wait for an existing instance to finish its current job.

Figure 5 presents queues that serve to provide messages from instances of module A to instances of module B. Note that module B is a simple module – it has only one single-token input. Thus, the incoming messages do not need to be grouped. However, we need to preserve the grouping made for module A at the start of the application. In step 1) we create a new queue family when a message arrives from one of the jobs being instances of module A. We note that this message has a stack with two layers. The first layer corresponds to the sequence numbering preserved from the tokens messages for token no. "1". This reflects a series of folders placed on the application input "Videos". The second layer contains a new sequence (a sub-sequence), corresponding to messages created from one instance of module A. These messages reflect individual files present in a given folder.

The first queue family is identified by the token number

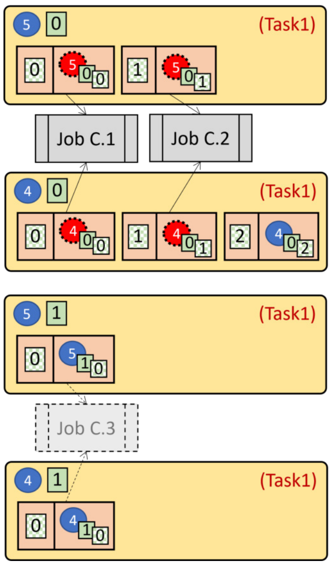


Fig. 6. Queues for tokens 4 and 5

and the top-level sequence number (here: 3-0). It has one default queue that does not need any additional identification. In Figure 5, we can see that the queue already has one job (B.1) assigned as its recipient and the first token message (3-0-0) is already being processed by this job. In step 2), another message in the sequence (3-0-1) is delivered to the queue. This creates an occasion to start another job of type "B" (assuming that job B.1 still processes token 3-0-0). Such a new job (here: B.2) would subscribe to the queue the same way as job B.1. In this case, further token messages can be processed by two jobs.

Step 3) in Figure 5 shows the configuration of queues after several other messages for token "3" arrive. As we can see, further queue families were created. These families reflect the token message sequence arriving at module A. For each such token message (1-0, 1-1, 1-2), we create a separate queue family (3-0, 3-1, 3-2). The reason for applying such a mechanism is to facilitate the distribution of computations. Each of the queues can be assigned to a different computation node potentially located in different geographical locations. In such a case, e.g. jobs for queue 3-0 will obviously not be able to process token messages in queue 3-1. On the other hand, if both queues are assigned to a single node, it is fairly easy to assign a job to both queues as their recipient.

Figure 6 presents queues that handle messages coming from instances of modules A and B and sent to instances of module C. The situation is to some extent similar to the previous ones, so we present only one "snapshot" of the queues. The main difference is the existence of many queues in queue families with split token sequence stacks. This split is caused to facilitate the distribution of jobs. Queue families are assigned to specific computation nodes, and individual queues in these families are assigned to specific computation module instances.

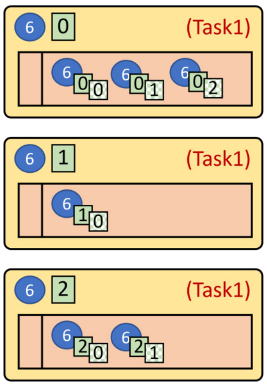


Fig. 7. Queues for token 6

In our example, we have two jobs already created based on two sets of token messages (5-0-0 with 4-0-0 and 5-0-1 with 4-0-1). These two jobs are running on one computation node. Another job (C.3) starts based on messages 5-1-0 and 4-1-0. This job can potentially be started on a different computation node.

Finally, Figure 7 shows queues containing token messages with the final results of the application. Here we can see three queue families created for a single token ("6"). These families correspond to the initial token message sequence shown in Figure 4. For instance, queue family 6-0 corresponds to tokens 1-0 and 2-0 sent initially to queues "1" and "2". As we remember, these initial tokens reflected folders with video and subtitle files. At this final stage, the results should also be grouped into folders according to the CAL application (see the output "Films" in Figure 1). Messages assigned to each of the queues in Figure 7 reflect individual files with the resulting films. Queues reflect folders into which these files should be sent. These output folders will contain films created from videos and subtitles contained in synchronised input folders.

Note that the application can be easily extended with further modules. In such a case, the queues for token "6" could be exactly the same (depending on how we exactly extend our application). In such a hypothetical case, the first arrival of a token message will create an occasion to start a new job, to which all messages from the given queue would be sent. It has to be stressed that our multi-queue mechanisms allow for the fully automatic creation of queues, as in the presented example. The execution engine can create queues based on the definition of the application in CAL and the arrival of consecutive token messages. In the next section, we present the implementation of these mechanisms.

V. Multi-queue implementation

The BalticLSC system consists of several components that cooperate to orchestrate jobs in a distributed computation environment. An extract of the BalticLSC logical architectural model (see https://www.balticlsc.eu/model/) is shown in Figure 8. The multi-queue mechanisms presented in this paper are

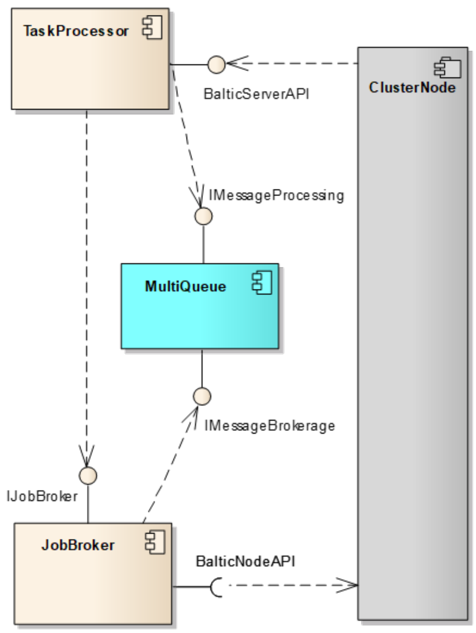


Fig. 8. Multi-queue usage in a computation system

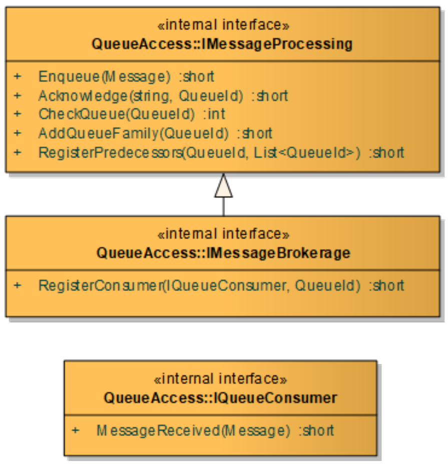

«internal interface»
QueueAccess::IMessageProcessing
+ Enqueue(Message) :short
+ Acknowledge(string, QueueId) :short
+ CheckQueue(QueueId) :int
+ AddQueueFamily(QueueId) :short
+ RegisterPredecessors(QueueId, List<QueueId>) :short
«internal interface»
QueueAccess::IMessageBrokerage
+ RegisterConsumer(IQueueConsumer, QueueId) :short
«internal interface»
QueueAccess::IQueueConsumer
+ MessageReceived(Message) :short

Fig. 9. Interfaces of the multi-queue implementation

implemented within a separate component – "MultiQueue". This component works in cooperation with the middleware responsible for orchestrating computations. As we can see in Figure 8, this middleware contains two components that work as intermediaries between the multi-queue system and the actual computation modules.

The "TaskProcessor" component is responsible for invoking most of the presented multi-queue mechanisms. It receives token messages from computation modules through the Baltic-ServerAPI. It then appends them with all the necessary information (including the sequence stacks) and inserts them into the multi-queue. It also checks all the conditions for starting new computation module instances. When a condition is met, it passes an appropriate command to the "JobBroker" component. This component is responsible for the actual initiation of new module instances, which includes making decisions regarding job balancing and sensing these jobs to specific computation nodes. It also registers these new instances in the queues, allowing appropriate tokens to be transmitted.

To communicate with the multi-queue, the Task Processor uses the IMessageProcessing interface, and the Job Broker uses the IMessageBrokerage interface. The details of these interfaces are shown in Figure 9. The first of the interfaces groups operations to enqueue new token messages, acknowledge the finishing of their processing and check the status of tokens in specific queues. For managerial purposes, there is also the possibility to create queue families. Finally, there is the possibility of forming a tree-like structure from the queues by defining queue predecessors. This additional mechanism is used by an automatic queue-cleaning mechanism. This ensures that empty queues will not be deleted as long as preceding queues have unacknowledged tokens left. The IMessageBrokerage interface contains an additional method that allows registering new recipients for specified queues. Figure 9 also shows the IQueueConsumer interface that should be implemented by a computation module. This simple interface allows module instances to receive messages.

Operation of the TaskProcessor component when processing a token message is presented in Figure 10. The whole process is initiated by invoking the PutTokenMessage method in reaction to the incoming token message. First, the method checks if there is the need to start a so-called job batch. This feature of the BalticLSC system allows for grouping of jobs so that they would be computed on the same computation nodes making data transfer more efficient (see CAL specifications [19]). Starting of job batches is similar to starting of jobs. The TaskProcessor needs to check all the queues related to the inputs of all jobs that are also inputs of the particular job batch. For this, it calls the CheckQueue operation which returns the number of messages in a given queue. Presence of at least one message in all the input queues required to start the job batch results in calling the ActivateJobBatch operation of the JobBroker component. Obviously, if the job batch is already activated, there is no need to perform queue checks and activate it again.

A similar approach is used to activate the specific job related to the processed token. Again, the procedure includes checking of all the required input queues. Note that the queue for the processed token message does not need to be checked. Instead, the message is enqueued after all the necessary checks are done.

Details of the multi-queue implementation are shown in Figure 11. The MultiQueue class is responsible for the functioning of the MultiQueue component and implements its provided interfaces. It runs a thread (the "Run" operation) that periodically sends tokens to consumers registered with the various queues. Message distribution is performed through handles to queue families grouped by task identifiers. These queue families are implemented by the MsgQueueFamily

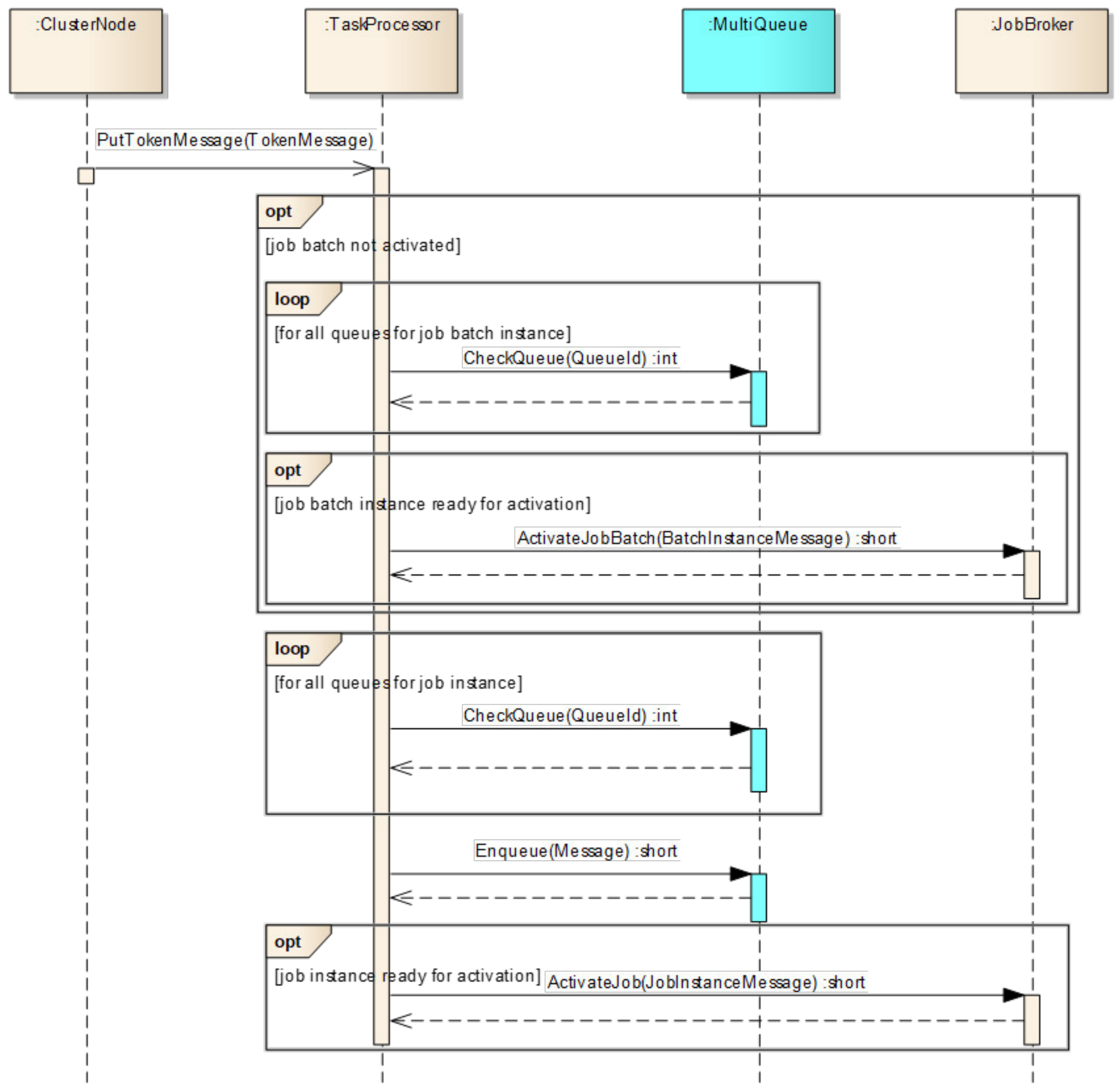

:ClusterNode
:TaskProcessor
:MultiQueue
:JobBroker
PutTokenMessage(TokenMessage)
opt
[job batch not activated]
loop
[for all queues for job batch instance]
CheckQueue(QueueId) :int
opt
[job batch instance ready for activation]
ActivateJobBatch(BatchInstanceMessage) :short
loop
[for all queues for job instance]
CheckQueue(QueueId) :int
Enqueue(Message) :short
opt
[job instance ready for activation]
ActivateJob(JobInstanceMessage) :short

Fig. 10. Sequence diagram illustrating processing of a token message

class, which groups queues with a common ‘name’. This name is formed using the same data structure (“QueueIdentifiable”) as in the original data model (Figure 2). The queue family class has various operations that allow to enqueue and acknowledge messages, distribute them to the registered consumers, and check queue statuses.

As expected, the MsgQueueFamily class contains the MsgQueue class. This class holds the actual queue contents, i.e. the messages. It also contains the name suffix in the form of a token sequence stack and has handles to the queue consumers. Operations of this class are typical for a queue and do not necessitate a detailed explanation.

To illustrate the usage of our multiqueue within the BalticLSC system we will use the CAL program fragment presented in Figure 12. The main module in this fragment is the “ImgChannelJoin” module. It has three input pins and one output pin. The three inputs are connected with certain processing modules, where one of them is shown in the figure. The input pins are associated with appropriate tokens (np. 3, 9 and 10). In the figure, we can see three tokens arriving, each of them having the same sequence number (0) and thus causing initiation of a new job instance for the “ImgChannelJoin” module.

In Figure 13 we can see a fragment of execution log for the situation in Figure 12. The log contains information about processing of a single token message arriving at one of the inputs of the “joiner” module. The first entry in the log denotes the arrival of the message at the Task Processor component (see also Figure 8). As we can see, the message is related to Token number 3. It is the first in a sequence of messages for this Token, and thus contains the index number 0 on its sequence stack (see the ‘seq_stack’ section).

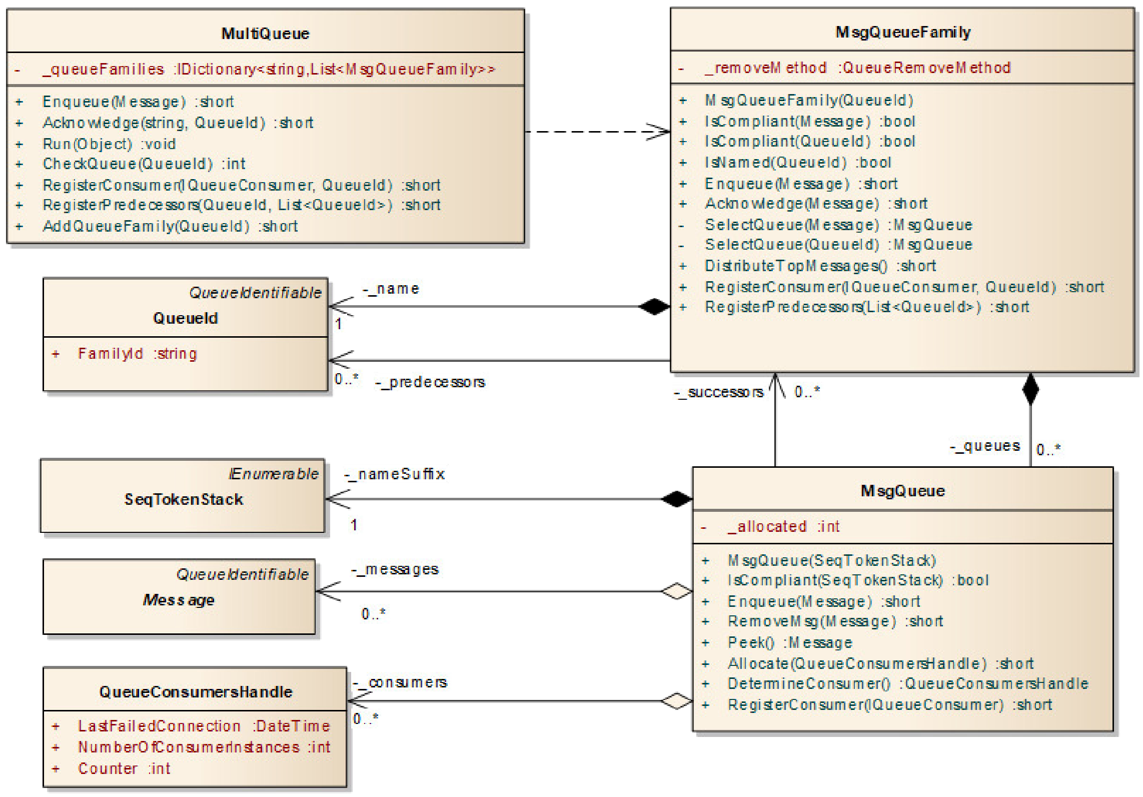

MultiQueue
- _queueFamilies :IDictionary<string,List<MsgQueueFamily>>
+ Enqueue(Message) :short
+ Acknowledge(string, QueueId) :short
+ Run(Object) :void
+ CheckQueue(QueueId) :int
+ RegisterConsumer(IQueueConsumer, QueueId) :short
+ RegisterPredecessors(QueueId, List<QueueId>) :short
+ AddQueueFamily(QueueId) :short
MsgQueueFamily
- _removeMethod :QueueRemoveMethod
+ MsgQueueFamily(QueueId)
+ IsCompliant(Message) :bool
+ IsCompliant(QueueId) :bool
+ IsNamed(QueueId) :bool
+ Enqueue(Message) :short
+ Acknowledge(Message) :short
- SelectQueue(Message) :MsgQueue
- SelectQueue(QueueId) :MsgQueue
+ DistributeTopMessages() :short
+ RegisterConsumer(IQueueConsumer, QueueId) :short
+ RegisterPredecessors(List<QueueId>) :short
QueueIdentifiable
QueueId
+ FamilyId :string
-_name
1
0..*
-_predecessors
-_successors
0..*
-_queues
0..*
IEnumerable
SeqTokenStack
-_nameSuffix
1
MsgQueue
- _allocated :int
+ MsgQueue(SeqTokenStack)
+ IsCompliant(SeqTokenStack) :bool
+ Enqueue(Message) :short
+ RemoveMsg(Message) :short
+ Peek() :Message
+ Allocate(QueueConsumersHandle) :short
+ DetermineConsumer() :QueueConsumersHandle
+ RegisterConsumer(IQueueConsumer) :short
QueueIdentifiable
Message
-_messages
0..*
QueueConsumersHandle
+ LastFailedConnection :DateTime
+ NumberOfConsumerInstances :int
+ Counter :int
-_consumers
0..*

Fig. 11. Code structure of the multi-queue implementation

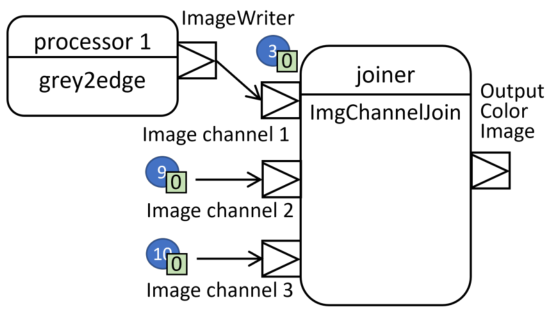


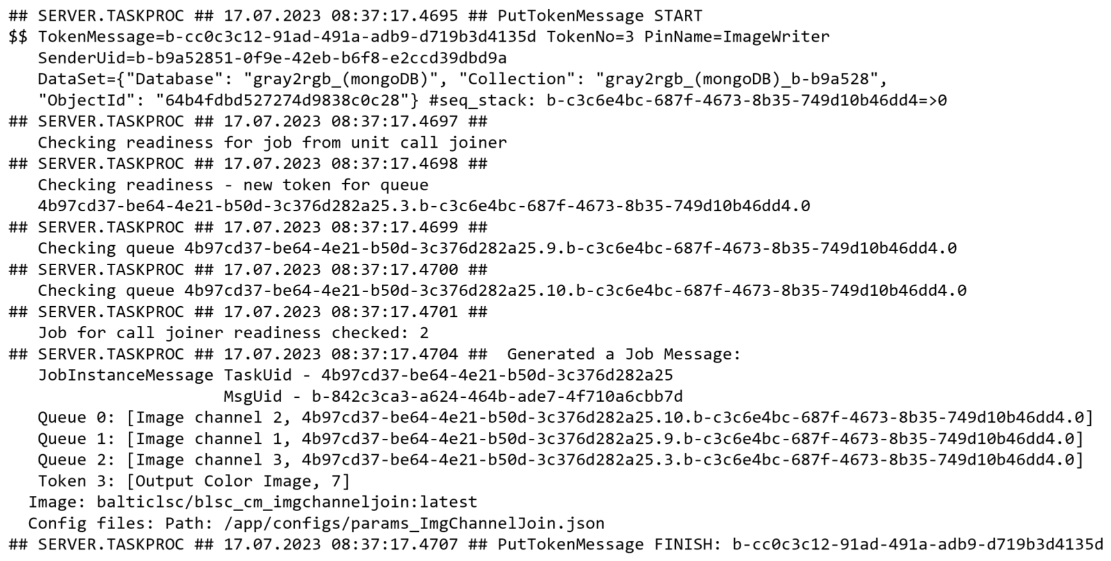
Fig. 12. CAL program fragment to illustrate queue module usage

Immediately after receiving the message, the Task Processor performs a sequence of checks to determine if the incoming message would trigger a new job instance (see the "CheckQueue" operation). In our case, three queues are checked – the ones related to the three inputs of the "joiner" module, including the queue for the incoming message. As we can see, the Task Processor requests checking queues for Tokens 3, 9 and 10 with the sequence index 0.

In the presented situation, queues "9.0" and "10.0" already had messages in them. This means that arrival of the message to the queue "3.0" should trigger a new job instance. The log in Figure 13 thus contains information about creating a new job message. This new message contains information about the queues for input tokens and about the output token. It also determines technical details of the module to be executed by the runtime. Some of these details (image file, configuration file) are shown in the figure but other are omitted as not relevant here. Note that the information about the module to be executed are determined by the Task Processor based on information compiled from the appropriate CAL program.

The new job message is passed to the "JobBroker" component through the "IJobBroker" interface (see again Figure 8). The broker then selects an appropriate computation node for execution of the job and sends the job message to this node. Note that job brokerage algorithm is out of scope of this paper. After determining the job's executing module at the computation node, the broker registers it in all the relevant queues through the "IMessageBrokerage" interface. Finally, the Task Processor inserts the incoming token message into the multi-queue using the "Enqueue" operation of the "IMessageProcessing" interface (see Figure 9). This finishes processing of the token message by the Task Processor. This is signalled by the last entry in the presented log. Further steps involve the computation node that accesses the queues according to the description in the previous sections.

By examining the log in Figure 13 we can also determine the efficiency of the queue system. Each entry contains information about its time with the precision of tenths of milliseconds. As we can see, it took the engine around a millisecond to process the message – put it into the queue module, check



```
## SERVER.TASKPROC ## 17.07.2023 08:37:17.4695 ## PutTokenMessage START
$$ TokenMessage=b-cc0c3c12-91ad-491a-adb9-d719b3d4135d TokenNo=3 PinName=ImageWriter
   SenderUid=b-b9a52851-0f9e-42eb-b6f8-e2ccd39dbd9a
   DataSet={"Database": "gray2rgb_(mongoDB)", "Collection": "gray2rgb_(mongoDB)_b-b9a528",
   "ObjectId": "64b4fdbd527274d9838c0c28"} #seq_stack: b-c3c6e4bc-687f-4673-8b35-749d10b46dd4=>0
## SERVER.TASKPROC ## 17.07.2023 08:37:17.4697 ##
   Checking readiness for job from unit call joiner
## SERVER.TASKPROC ## 17.07.2023 08:37:17.4698 ##
   Checking readiness - new token for queue
   4b97cd37-be64-4e21-b50d-3c376d282a25.3.b-c3c6e4bc-687f-4673-8b35-749d10b46dd4.0
## SERVER.TASKPROC ## 17.07.2023 08:37:17.4699 ##
   Checking queue 4b97cd37-be64-4e21-b50d-3c376d282a25.9.b-c3c6e4bc-687f-4673-8b35-749d10b46dd4.0
## SERVER.TASKPROC ## 17.07.2023 08:37:17.4700 ##
   Checking queue 4b97cd37-be64-4e21-b50d-3c376d282a25.10.b-c3c6e4bc-687f-4673-8b35-749d10b46dd4.0
## SERVER.TASKPROC ## 17.07.2023 08:37:17.4701 ##
   Job for call joiner readiness checked: 2
## SERVER.TASKPROC ## 17.07.2023 08:37:17.4704 ##  Generated a Job Message:
   JobInstanceMessage TaskUid - 4b97cd37-be64-4e21-b50d-3c376d282a25
                      MsgUid - b-842c3ca3-a624-464b-ade7-4f710a6cbb7d
   Queue 0: [Image channel 2, 4b97cd37-be64-4e21-b50d-3c376d282a25.10.b-c3c6e4bc-687f-4673-8b35-749d10b46dd4.0]
   Queue 1: [Image channel 1, 4b97cd37-be64-4e21-b50d-3c376d282a25.9.b-c3c6e4bc-687f-4673-8b35-749d10b46dd4.0]
   Queue 2: [Image channel 3, 4b97cd37-be64-4e21-b50d-3c376d282a25.3.b-c3c6e4bc-687f-4673-8b35-749d10b46dd4.0]
   Token 3: [Output Color Image, 7]
  Image: balticlsc/blsc_cm_imgchanneljoin:latest
  Config files: Path: /app/configs/params_ImgChannelJoin.json
## SERVER.TASKPROC ## 17.07.2023 08:37:17.4707 ## PutTokenMessage FINISH: b-cc0c3c12-91ad-491a-adb9-d719b3d4135d
```

Fig. 13. CAL program execution log fragment

associated queues and create a job message. This time is negligible in relation to the time used for computations which usually takes minutes, hours or even days. It thus can be argued that such message processing times allow for significant scalability of the system. Even with a single instance of the runtime engine, many parallel tasks could be processed without noticeable delays related to message processing. If needed, this could be further improved by adding additional runtime engine instances running on several machines. Note that determining detailed performance characteristics of our solution are subject to future work.

An important additional feature worth noting as facilitated by the existence of queue families is "queue garbage collection". Each queue has information about its successors, and each queue family has information about its predecessors. This allows the multi-queue to keep track of queues that are not used and will not be used in the future (within the current application instance). The garbage collector mechanism traverses queues and queue families to determine all the relations for a given queue and based on an appropriate algorithm removes "dead" queues. Discussion on the details of this mechanism is out of the scope of this paper.

VI. Conclusion

The presented multi-queue concept was validated in a fully operational distributed computation system. It has proven to be effective means of orchestrating computations with diverse configurations of data sets, flowing between instances of computation modules. The users of the BalticLSC system have developed various computation applications with different configurations of data flows. Examining of computation logs from the system acknowledges the effectiveness of message distribution by the queue system. Message handling times are short (in the scale of milliseconds) and thus negligible within the whole computation process.

The characteristics of our multi-queues facilitate the simultaneous delivery of multiple token messages to many instances of the same computation module running on different computation nodes. It also allowed the implementation of a job initiation mechanism that is based on the availability of data transmitted through complex data processing paths. At the same time, the implementation of our multi-queue system is quite simple and results in a lightweight queue component that can be used in similar contexts.

References

[1] Adam Barker, Paolo Besana, David Robertson, and Jon B. Weissman. The benefits of service choreography for data-intensive computing. In *Proceedings of the 7th International Workshop on Challenges of Large Applications in Distributed Environments*, CLADE '09, page 1–10, New York, NY, USA, 2009. Association for Computing Machinery.

[2] Li Chunlin, Tang Jianhang, and Luo Youlong. Multi-queue scheduling of heterogeneous jobs in hybrid geo-distributed cloud environment. *The Journal of Supercomputing*, 74:5263–5292, 2018.

[3] Philippe Dobbelaere and Kyumars Sheykh Esmaili. Kafka versus RabbitMQ: A comparative study of two industry reference publish/subscribe implementations: Industry paper. In *Proceedings of the 11th ACM International Conference on Distributed and Event-Based Systems*, DEBS '17, page 227–238, New York, NY, USA, 2017. Association for Computing Machinery.

[4] Nishant Garg. *Apache Kafka*. Packt Publishing Birmingham, UK, 2013.

[5] Mohammad Hedayati, Kai Shen, Michael L Scott, and Mike Marty. Multi-queue fair queuing. In *USENIX Annual Technical Conference*, pages 301–314, 2019.

[6] Péter Kacsuk, editor. *Science Gateways for Distributed Computing Infrastructures*. Springer International Publishing, 2014.

[7] Peter Kacsuk, Zoltan Farkas, Miklos Kozlovszky, Gabor Hermann, Akos Balasko, Krisztian Karoczkai, and Istvan Marton. WS-PGRADE/gUSE generic DCI gateway framework for a large variety of user communities. *Journal of Grid Computing*, 10(4):601–630, nov 2012.

[8] Peter Kacsuk, József Kovács, and Zoltán Farkas. The Flowbster cloud-oriented workflow system to process large scientific data sets. *Journal of Grid Computing*, 16(1):55–83, jan 2018.

[9] A V Karthick, E Ramaraj, and R Ganapathy Subramanian. An efficient multi queue job scheduling for cloud computing. In *2014 World Congress on Computing and Communication Technologies*, pages 164–166, 2014.

[10] Haopeng Li and Hui Li. A scheduling strategy based on multi-queues of cassandra. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 2664–2669, 2017.

[11] Ji Liu, Esther Pacitti, Patrick Valduriez, and Marta Mattoso. A survey of data-intensive scientific workflow management. *Journal of Grid Computing*, 13:457–493, 2015.

[12] Pedro García López, Aitor Arjona, Josep Sampé, Aleksander Slominski, and Lionel Villard. Triggerflow: Trigger-based orchestration of serverless workflows. In *Proceedings of the 14th ACM International Conference on Distributed and Event-Based Systems*, DEBS '20, page 3–14, New York, NY, USA, 2020. Association for Computing Machinery.

[13] Krzysztof Marek, Michał Śmiałek, Kamil Rybiński, Radosław Roszczyk, and Marek Wdowiak. BalticLSC: Low-code software development platform for large scale computations. *Computing and Informatics*, 40(4):734–753, 2021.

[14] Chris Peltz. Web services orchestration and choreography. *Computer*, 36(10):46–52, 2003.

[15] Anastasiia Postnikova, Nikita Koval, Giorgi Nadiradze, and Dan Alistarh. Multi-queues can be state-of-the-art priority schedulers. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 353–367, 2022.

[16] Stephen Ross-Talbot. Orchestration and choreography: Standards, tools and technologies for distributed workflows. In *NETTAB Workshop-Workflows management: new abilities for the biological information overflow*, volume 1, page 8, Naples, Italy, 2005.

[17] Maciej Rostanski, Krzysztof Grochla, and Aleksander Seman. Evaluation of highly available and fault-tolerant middleware clustered architectures using RabbitMQ. In *2014 Federated Conference on Computer Science and Information Systems*, pages 879–884, 2014.

[18] Radoslaw Roszczyk, Marek Wdowiak, Michal Smialek, Kamil Rybinski, and Krzysztof Marek. BalticLSC: A low-code HPC platform for small and medium research teams. In *2021 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, oct 2021.

[19] Kamil Rybiński, Michał Śmiałek, Agris Sostaks, Krzysztof Marek, Radosław Roszczyk, and Marek Wdowiak. Visual low-code language for orchestrating large-scale distributed computing. *Journal of Grid Computing*, 21(3), jul 2023.

[20] Gaurav Sharma, Neha Miglani, and Ajay Kumar. PLB: a resilient and adaptive task scheduling scheme based on multi-queues for cloud environment. *Cluster Computing*, 24(3):2615–2637, 2021.

[21] T Sharvari and Nag K Sowmya. A study on modern messaging systems-Kafka, RabbitMQ and NATS streaming. 2019.

[22] Jaspreet Singh and Deepali Gupta. An smarter multi queue job scheduling policy for cloud computing. *International Journal of Applied Engineering Research*, 12(9):1929–1934, 2017.

[23] John Vineet and Liu Xia. A survey of distributed message broker queues. 2017.

[24] Steve Vinoski. Advanced Message Queuing Protocol. *IEEE Internet Computing*, 10(6):87–89, 2006.

[25] Guozhang Wang, Joel Koshy, Sriram Subramanian, Kartik Paramasivam, Mammad Zadeh, Neha Narkhede, Jun Rao, Jay Kreps, and Joe Stein. Building a replicated logging system with Apache Kafka. *Proceedings of the VLDB Endowment*, 8(12):1654–1655, aug 2015.