Permutable Queue as a 1-in-N-out Message Router

Gongzhu Hu
Department of Computer Science
Central Michigan University
Mount Pleasant, Michigan, USA
hulg@cmich.edu

Abstract
The basic queue is a first-in-first-out linear data structure (software component). It can be considered as a communication device with one input port and one output port. In the paper, we investigate the behavior of an extended queue that has one input channel and multiple output channels. An enter-queue command enters a data item into the queue as does an ordinary queue, but a de-queue command removes the front item and sends it to a specific output channel. This is similar to the function of a simplified message router. We define a set of stream functions to map between input stream and output stream to describe the behavior of the queue. Because of the items in the input stream may be re-ordered to meet the “routing” requirements, we call the software component “permutable queue.”

Keywords Software component specification, interactive queue, stream function, message routing

1. Introduction

The classic data structures (list, queue, stack, tree, etc.) are commonly used as software components in system and application developments. They have well-defined properties to support certain operations.

A basic queue is a linear data structure that supports operations enter-queue and de-queue for data items to “flow through”. That is, it has an input channel where data items are entered into the queue and an output channel where data items are removed from the queue. It is known as a first-in-first-out (FIFO) structure.

Many applications use queues as a communication channel between two parties that one party sends messages, each of which is composed of a sequence of packets, to the other party. The order in which the packets are received is the same as the order they were sent, which is exactly the property of a queue. There are many applications, however, that involve multiple receivers at the output-end of the queue. In order words, these applications need a “queue” with multiple output channels. It is not a queue in the regular sense; rather, it is an extended version of the basic queue with certain restrictions or variations to the queue operations. The primary operations on this kind of extended queues is still FIFO for data flow, but the enter-queue and de-queue operations are somewhat different from the ones for the basic queue. One simple example of such extended queue is a “split queue” that sends the incoming packets to each of the two or more output channels, like the electronic split device for telephone lines that fans-out the input signals into multiple lines. Another example is a message router that delivers incoming messages to different receivers.

We consider a queue that can be used for a simplified version of a message router, in which all senders send messages to the same input channel and the queue has a separate output channel dedicated for each receiver. It is a 1-in-N-out queue with the same enter-queue operation as regular queue but a different de-queue operation. When a de-queue operation is to be performed, the extended queue checks if the packet at the front of the queue, say p, is ready to be delivered to the target output channel. The packet p is sent to the output channel only if the channel is empty or has been used to deliver previous packets of the same message p belongs to. Otherwise, the enter-queue command is delayed. The postponement is done by moving the packet p from the front to a later position in the queue. In other words, the items in the queue is “shuffled” or “permuted” to an appropriate order, hence we call it permutable queue.

The main contribution of this paper is a formal specification of the behavior of the permutable queue as a software component using stream functions.
2. Related Work

Message router is a system component that has M input channels and N output channels representing N receivers. Messages are sent by senders through the router to the intended receivers. It’s up to the router to make the decision on how to deliver the individual packets in the messages to the destinations. A lot of research has been done on routers in the computer network field, but research on interactive routers as a software component specified in terms of stream processing functions is still limited. Creveul and Roman [2] gave a formal specification of a message router using eleven properties to satisfy the requirements of a router. A series of six refinements were defined that are used to lead to a design and construction of such a router. Cunningham and Cai [3] also proposed a specification of message router with 27 properties and treated it as an open system, i.e. the router interacts with an arbitrary environment.

Formal methods for system component specification have been studied and proposed in the literature, such as interactive model with asynchronous unidirectional channels [1] and communication history model [8]. In some of these studies, interactive components are often described using stream functions [9]. Dosch and Stümpel [7] discussed using state transition machine for the process from input stream to output stream. An abstract model using this basic approach for an interactive stack was proposed in [4] and for an interactive queue in [6].

A recent paper [5] also studies a 1-in-N-out component (called a distributor) for message routing. It uses a stream of “control numbers” to control the distribution of the input messages to the output channels, and defines a state transition machine as an implementation model. The approach presented in this paper is different from [5] mainly in message composition and the control mechanism. In our approach, there is no “control numbers” to direct messages to output channels, rather, the control is embedded in the packets of the massages.

We shall use the notions similar to the ones in [6] for the specification of permutable queues. The main difference is that we are dealing with a queue with multiple (rather than just one) output channels and the packets in the input stream are targeted to particular output channels.

3. Messages and Packets

A message is a data stream sent by a sender to a receiver. It consists of a sequence of N packets \(p_i\) in a total order. There are three types of packets: header, body, and tail. We use \(h\), \(b\), and \(t\), respectively, to denote the type of a packet. A message is identified by a massage id that consists of a message number (msgNo) and a receiver number (recNo). We may also use the term output channel number (outChNo) for recNo. Each message entered to the input channel has a unique message number and each output channel is associated with a unique receiver number. A message contains exactly one header and one tail, and zero or more (but finite) body packets in between. The header is the first and the tail is the last packet in a message. That is, a message can be expressed as

\[ M = \langle (\text{msgNo}, \text{recNo}), (p^h_0, p^b_1, \ldots, p^b_N, p'^{t}_{N+1}) \rangle \]

where the \(p^h_0\) is the header, the \(p'^{t}_{N+1}\) is the tail, and the body packets \(p^b_k\), \(k = 1, \ldots, N\), form a total order. That is, for any two body packets \(p^b_i\) and \(p^b_j\) in a message, \(p^b_i\) is entered to the queue before \(p^b_j\) if \(i < j\).

A message \(M\) is entered into the queue as a sequence of packets, and the message id is not a packet. Hence each packet in \(M\) also needs to include the message id. In addition to the data value, a packet also contains a packet number (pkcNo) as denoted in the subscript in the above notation to indicate the order in which the packet is entered to the queue. A packet in message \(M\) is expressed as

\[ p^x_k = \langle \text{msgNo}, \text{recNo}, x, k, d \rangle \]

where \(x\) is the packet type, \(k\) is the packet number, and \(d\) is the data value if \(x = b\). So, the header, tail, and body packets are specified as

\[
\begin{align*}
    p^h_0 &= \langle \text{msgNo}, \text{recNo}, h, 0 \rangle \\
    p'^{t}_{N+1} &= \langle \text{msgNo}, \text{recNo}, t, N + 1 \rangle \\
    p^b_k &= \langle \text{msgNo}, \text{recNo}, b, k, d \rangle, k \in [1, N]
\end{align*}
\]

To simplify the notation, we drop the packet numbers (0 and \(N + 1\)) for the header and tail packets because they are always at the two ends of a message, and drop the type \(b\) and use subscript for the packet number for body packets:

\[
\begin{align*}
    p^h_0 &= \langle \text{msgNo}, \text{recNo}, h \rangle \\
    p'^{t}_{N+1} &= \langle \text{msgNo}, \text{recNo}, t \rangle \\
    p^b_k &= \langle \text{msgNo}, \text{recNo}, d_k \rangle, k \in [1, N]
\end{align*}
\]

For example, message 1 targeted to receiver 2 with data values \(d_1, \cdots, d_N\) will be

\[ \langle (1, 2, h), (1, 2, d_1), \cdots (1, 2, d_N), (1, 2, t) \rangle \]

We may further simplify for the discussion in later sections by using the following symbols for packets:
header packet of message $i$ sent to receiver $j$.

tail packet of message $i$ sent to receiver $j$.

the $k$-th body packet of message $i$ with value $d$ sent to receiver $j$. We may drop $k$ if it causes no confusion.

We assume that the packets in a message are indeed entered into the queue in the right order.

4. Input and Output Streams

The input to the permutable queue is a sequence of enter-queue and de-queue commands, denoted $enq(p)$ and $deq$, respectively, where $p$ is a packet as one of the three types above. Let $\mathcal{P}$ be the domain of packets $p$. The input is defined as

$$I = enq(\mathcal{P}) \cup \{deq\}$$

Each output channel $i$ ($i \in [1,N]$) receives a packet, or no packet at all when no packet is ready for the channel. The output of the queue (to each channel) is defined as

$$O = \mathcal{P} \cup \{nopacket\}$$

A $deq$ command from the input stream will cause the queue to deliver a packet (if any) to the proper output channel only if certain requirements (discussed in the next section) are met. Some of these requirements involve the current status of the target output channel. Hence, the queue with $N$ output channels is a stream function that maps (input stream, output stream) to output stream:

$$queue : I^* \times O^* \rightarrow O^*$$

Note that the permutable queue differs from regular queue in the way the output is handled. It needs to guarantee the ordering of the messages at each output channel.

5. Behavior of Permutable Queue

The permutable queue models the 1-in-N-out message router where the relationship between senders and receivers is 1:N. That is, a sender can send messages to one or more receivers.

5.1. Requirements

The behavior of the queue needs to satisfy the following requirements, some of which were given in [2].

R1. Each message starts with a header packet and ends with a tail packet.

R2. The ordering of the packets in a message is preserved.

R3. Messages going to the same output channel are not interleaved.

R4. All packets are eventually de-queued.

The first requirement R1 indicates that the length of a message is finite and the two special packets (header and tail) define its boundaries. R2 requires that the order in which the packets within a message are entered to the queue be the same as the order they are de-queued. R3 states that from the receiver’s point of view, it receives messages rather than individual packets. That is, interleaving packets of different messages entered to the input channel targeted to the same output channel must be serialized. Requirement R4 says that the number of de-queue commands must be no less than the enter-queue commands.

5.2. An Example

Let’s look at an example of a permutable queue with two output channels. Three messages with message id’s $(1,2), (2,2)$ and $(3,1)$ are entered to the queue in the order given in an interleaving manner. Each packet $p_i$ is entered via an $enq(p_i)$ command in the input stream. In the following we use $\langle \cdots \rangle$ to denote a sequence of packets. Here is a sequence of packets entered to the queue, in that order:

$$\langle(2,2,h), (2,2,a_1), (3,1,h), (2,2,a_2), (1,2,h), (1,2,b_1), (3,1,c_1), (1,2,b_2), (3,1,c_2), (1,2,b), (3,1,c_3), (2,2,b), (3,1,t) \rangle$$

These packets are sent to the output channels with $deq$ commands in the input stream. Although the order of the $deq$ commands in relations to the $enq$ commands in the input stream does have impact on the output, we assume that there are enough $deq$ commands to flush all packets to the output channels.

The messages 1 and 2 are sent to the same output channel in the order of the arrival of their header packets. Since the header of message 2 is entered before the header of message 1, the output channel 2 will receive all the packets of message 2 before any packet of message 1. In other words, the interleaving packets of message 1 and 2 are serialized. Message 3 is sent to output channel 1. The packets are delivered to the output channels via a $deq$ command in the input stream. The two output channels will receive packets:

$$chl 1 : \langle(3,1,h), (3,1,c_1), (3,1,c_2), (3,1,c_3), (3,1,t) \rangle$$

$$chl 2 : \langle(2,2,h), (2,2,a_1), (2,2,a_2), (2,2,t) \rangle,$$

$$\langle(1,2,h), (1,2,b_1), (1,2,b_2), (1,2,t) \rangle$$
We can see that the \textit{enq} and \textit{deq} commands interleaving in an input stream is permuted so that the messages are serialized at the output channels.

5.3. Specification of Queue Stream Function

We adopt the notion similar to [6]. We use capital letters like \(X, Y\) to represent a stream (e.g. \(X = \langle x_1, \cdots, x_n \rangle\) may be a sequence of \textit{enq} and \textit{deq} commands in the input stream, or a sequence of packets in an output stream). The symbol \& is used as the concatenation operator on two streams. That is, for \(X = \langle x_1, \cdots, x_n \rangle\) and \(Y = \langle y_1, \cdots, y_m \rangle\), the concatenation of \(X\) and \(Y\) is \(X \& Y = \langle x_1, \cdots, x_n, y_1, \cdots, y_m \rangle\). We also use \textit{Enq} and \textit{Deq} to represent zero or more \textit{enq} and \textit{deq} commands, respectively:

\[
\text{Enq} \in \text{enq}(\mathcal{P})^*, \text{Deq} \in \text{deq}^*
\]

Let \(Out\) be the current output stream, which is also expressed as \(\langle \cdot \cdot \cdot \rangle\) to represent the output stream at a particular channel.

The behavior of the permutator is specified by defining the stream functions. First, some basic ones:

\[
\begin{align*}
\text{queue}(\text{Enq}, \langle \rangle) &= \langle \rangle & (1) \\
\text{queue}(\text{Deq} \& X, \langle \rangle) &= \text{queue}(X, \langle \rangle) & (2) \\
\text{queue}(\text{Enq}, \text{Out}) &= \text{Out} & (3) \\
\text{queue}(\text{Deq} \& X, \text{Out}) &= \text{queue}(X, \text{Out}) & (4)
\end{align*}
\]

Enter-queue commands without \textit{deq} does not produce any additional output (Equation 3), i.e. the output stream stays the same as before. De-queue commands without prior \textit{enq} produces no data and leave the remaining input stream unchanged (Equation 4). Equation 1 and 2 are special cases where the output channels are empty. Another version of Equation 4 is to postpone the \textit{deq} commands. One way of achieving this is to process \(X\) before \textit{Deq}:

\[
\text{queue}(\text{Deq} \& X, \text{Out}) = \text{queue}(X \& \text{Deq}, \text{Out}) \quad (4')
\]

For \textit{deq} commands to produce output to a particular channel, it must be preceded by at least an \textit{enq} command that satisfies certain conditions on the current output stream regarding to the intended output channel. Hence:

\[
\begin{align*}
\text{queue}(\langle \text{enq}(p^i_j) \rangle \& \text{Enq} \& \langle \text{deq} \rangle \& X, \langle \cdot \cdot \cdot o^k_{jm} \rangle) &= \langle p^i_j \rangle \& \text{queue}(\text{Enq} \& X, \langle \cdot \cdot \cdot o^k_{mj} \rangle), \\
& \text{if} \ (o^k_{jm} = \text{tail}) \lor (k = i)
\end{align*}
\]

Equation (5) indicates that a packet \(p^i_j\) in message \(i\) sending to receiver \(j\) entered into the queue is later dequeued to the output channel \(j\) only if the last packet \(o^k_{jm}\) (the most recently de-queued to channel \(j\)) is a tail so that the channel has received a message in its entirety and is ready to accept a new message, or is a packet in the same message of \(p^i_j\) (i.e. \(k = i\)). In the later case, \(p^i_j\) is the packet immediately following \(o^k_{jm}\) in the ordering of the message. That is, \(p^i_j\) has an packet order number \(m + 1\).

When the condition in Equation (5) is not met, \(p^i_j\) is a packet that is in a message different from \(o^k_{jm}\) and \(o^k_{jm}\) is not a tail packet. In this case, the \textit{enq}(\(p^i_j\)) command has to be postponed to a later time, which is after all \textit{enq}(\(p^i_j\)) commands in the input stream and just before the next packet in the same message (\(i, j\)) is en-queue. Namely, the packet \(p^i_j\) is held until all packets in the current message \((k, j)\) has de-queued to the output channel \(j\). Let

\[
\text{Enq} = \text{enq}(d^i_{s_1}), \cdots, \text{enq}(d^i_{s_m})
\]

where the \(d\)'s are packets in Equation(5). If none of the \(d\)'s is from the same message of \(p^i_j\), the \textit{enq}(\(p^i_j\)) command is postponed to after the next \textit{deq} command:

\[
\text{queue}(\langle \text{enq}(p^i_j) \rangle \& \text{Enq} \& \langle \text{deq} \rangle \& X, \langle \cdot \cdot \cdot o^k_{jm} \rangle)
\]

\[
= \text{queue}(\text{Enq} \& \langle \text{deq} \rangle \& (\text{enq}(p^i_j)) \& X, \langle \cdot \cdot \cdot o^k_{mj} \rangle),
\]

\[
\text{if} \ (r_i \neq i, \ l \in [1, m])
\]

Otherwise, \textit{enq}(\(p^i_j\)) is merged with \textit{Enq}:

\[
\text{queue}(\langle \text{enq}(p^i_j) \rangle \& \text{Enq} \& \langle \text{deq} \rangle \& X, \langle \cdot \cdot \cdot o^k_{jm} \rangle)
\]

\[
= \text{queue}(\text{Enq} \& \langle \text{deq} \rangle \& (\text{enq}(p^i_j)) \& X, \langle \cdot \cdot \cdot o^k_{mj} \rangle),
\]

\[
\text{if} \ (o^k_{jm} \neq \text{tail}) \land (k \neq i)
\]

where \(\text{Enq}'\) is \textit{Enq} with \langle \text{enq}(p^i_j) \rangle inserted at the right position:

\[
\text{Enq}' = \langle \text{enq}(d^i_{s_1}), \cdots, \text{enq}(d^i_{s_i}), \text{enq}(p^i_j), \text{enq}(d^i_{s_{i+1}}), \cdots, \text{enq}(d^i_{s_m}) \rangle
\]

where \((r_1, \cdots, r_i) \neq i \land ((r_{i+1} = i) \land (s_{i+1} = j))\).

Equations (7) and (8) show where permutation occurs in the input stream to “sort out” the order of the \textit{enq} commands so that the next \textit{deq} can work properly. Note that we have assumed that the packets within the same message are entered to the queue in the right order, so that the packet \(d^i_{s_{i+1}}\) is the one immediately following \(p^i_j\). That is, \textit{enq}(\(p^i_j\)) is postponed to the appropriate position.

Finally, all the packets in the input are eventually delivered to the output channels. That is, for each \textit{enq}(\(p^i_j\)) there is a \textit{deq} that appears later in the input stream to de-queue \(p^i_j\) to the output channel \(j\) \((j = 1, \cdots, N)\).
6. Implementation

There are several ways to describe a possible implementation. One of the ways is to use a state transition machine that defines a transition function of the state of the queue upon each input:

$$\delta : Q \times I \rightarrow Q \times O$$

where $Q$ is a finite set of states. This approach has been used for implementing several software components [4, 6, 7]. For the permutable queue, we simply use some internal data structures and a procedural algorithm, and leave the use of a state transition machine as a future research topic.

6.1. Data Structures

Since the $enq$ commands in the input stream are targeted to specific output channels, and the messages for each channel are “sorted” based on the “timestamps” of the header packets of the messages, we can create a regular queue (one input and one output) to temporarily hold the packets in each message that are yet to be delivered to the output channel. The data structures are illustrated in Figure 1.

6.2. Algorithm

Initially each output channel $j$ is empty. The packets of the earliest arriving message $M$ are sent to the target channel. Packets in the other messages targeted to the same channel $j$ are entered into the internal queues. Upon the last packet (a tail) of the message $M$ have been delivered to the channel, the packets of the next message (its header packet arrived the earliest) are sent to channel $j$. The algorithm outline is given below in Algorithm 1.

**Algorithm 1 Permutable queue mapping: Input $\rightarrow$ Output**

1: Let output channel be $O_j, j \in [1,N]$. Input stream be $I$.
2: Let the internal queues be $Q_{jl}, j \in [1,N]$ for message $l$ targeted for output channel $j$. Each queue is associated with a timestamp value.
3: Initialization: All $Q_{jl}$ empty; $l = 0$; $timestamp = 0$; numDeq = 0.
4: loop
5: Let $c$ be the next item in the input stream $I$.
6: if $c = enq(h_j^i)$, i.e. a header packet then
7: Process entering a header packet $h_j^i$.
8: else if $c = enq(t_j^i)$, i.e. a tail packet then
9: Enter a packet ($t_j^i$).
10: else if $c = enq(p_j^i)$, i.e. a data packet then
11: Enter a packet ($p_j^i$).
12: else
13: Process a de-queue command.
14: end if
15: end loop

The following are the details for handling different commands. Algorithm 2 is for the $enq(h_j^i)$ command. Basically, a new internal queue for channel $j$ is created and associated it with the current $timestamp$ if output channel $O_j$ is holding packets of a message. The queue...
number of the next internal queue of the same output channel is incremented by 1.

**Algorithm 2** Process entering a header packet \( c = \text{enq}(h_j) \)

1: if \( O_j \) is empty or the latest packet in \( O_j \) is a tail
2: Send \( h_j \) to \( O_j \).
3: else
4: Create an internal queue \( Q_{ji} \) associated with output channel \( j \).
5: timestamp of \( Q_{ji} \leftarrow \text{timestamp} \)
6: Enter \( h_j \) to \( Q_{ji} \).
7: \( l \leftarrow l + 1 \)
8: \( \text{timestamp} \leftarrow \text{timestamp} + 1 \)
9: end if

Algorithm 3 enters a packet to the appropriate internal queue. A de-queue operation is performed if there is a delayed de-queue command.

**Algorithm 3** Enter a packet \( c = \text{enq}(p_i) \)

1: Enter \( p_i \) to \( Q_{ji} \).
2: if \( \text{numDeq} > 0 \) then
3: Process a de-queue command.
4: \( \text{numDeq} \leftarrow \text{numDeq} - 1 \)
5: end if

Process of de-queue command is shown in Algorithm 4. The internal queue with the earliest timestamp is de-queued if the queue is not empty. Otherwise the de-queue command is postponed.

**Algorithm 4** Process a de-queue command \( c = \text{deq} \)

1: if \( l = 0 \), i.e. all internal queues are empty then
2: \( \text{numDeq} \leftarrow \text{numDeq} + 1 \) \{delay the de-queue\}
3: else
4: Find an internal queue \( Q_{av} \) with the smallest timestamp among the queues for output channel \( O_v \) that is receiving message \( v \).
5: if \( Q_{av} \) is not found then
6: \( \text{numDeq} \leftarrow \text{numDeq} + 1 \) \{delay the de-queue\}
7: else
8: De-queue \( Q_{av} \) and send the item to \( O_v \).
9: if the item just de-queued is a tail packet and \( Q_{av} \) becomes empty then
10: Dispose \( Q_{av} \).
11: end if
12: end if
13: end if

Line 4–6 is when there is no temporarily held packet that can be delivered to an output channel. The deq command needs to be postponed. If algorithm 4 was invoked by algorithm 3, the number of delayed deq’s is not increased simply because line 6 in algorithm 4 and line 4 of algorithm 3 cancel out with each other.

It is important to note that there may be different ways to handle the case where a deq command is encountered in the input stream but there is no packet ready to be de-queued (i.e., try to de-queue an empty queue). In the algorithms we described here the deq command is delayed until the next enq, as suggested in Equation (4′). This strategy may also be called a fault correcting queue.

### 7. Example

We shall use the packets in the previous example to illustrate the implementation, with enq and deq commands. Assume that message 3 to channel 1 is composed of data values \( \alpha, \beta, \cdots \). Values in message 1 to channel 2 are \( a, b, \cdots \), and message 2 to channel 2 are \( x, y, \cdots \). Using \( p_i^j \) for packet in message \( i \) to channel \( j \) with data \( p \), the input stream consists of the following commands:

- \( \text{enq}(h_2^j), \text{enq}(x_2^j), \text{deq}, \text{enq}(h_1^j), \text{deq}, \text{enq}(y_2^j), \text{deq}, \text{enq}(h_1^j), \text{deq}, \text{enq}(a_1^j), \text{deq}, \text{enq}(\alpha_1^j), \text{deq}, \text{enq}(x_1^j), \text{deq}, \text{enq}(h_2^j), \text{deq}, \text{enq}(y_1^j), \text{deq}, \text{deq} \)

After the eight commands in the first line have been processed, the four packets in the enq commands have been de-queued to the output channels. The packets \( h_2^2 \), \( x_2^2 \), and \( y_2^2 \) are in output channel 2 and \( h_1^1 \) is in output channel 1.

The next deq command (first in the second line) is delayed because no packet can be de-queued at this time. Upon the next enq command entering the packet \( h_1^2 \) (which is in message 1), the delayed deq command is invoked trying to deliver it to channel 2. However the packet has to be postponed because channel 2 is currently delivering message 2. Hence \( h_1^2 \) is temporarily held in the internal queue \( Q_{21} \) with timestamp = 1.

When the next three enq and three deq commands are processed, the packets \( a_2^1 \) and \( b_1^1 \) are also stored in the internal queue \( Q_{21} \) together with \( h_2^2 \) since they are from the same message, while \( Q_1^2 \) is sent to channel 1. The state of the queue at this time is shown in Figure 2.

In the figure, \( d \) donates a deq command and \( v_j \) is short for \( \text{enq}(v_j) \).

For the rest commands in the input stream, the
Figure 2. A snapshot of the queue

deq's will send the packets as follows:

\[ \beta^3_1 \rightarrow \text{output channel 1} \]
\[ t^2_1 \rightarrow \text{internal queue} \]
\[ \gamma^3_1 \rightarrow \text{output channel 1} \]
\[ t^2_2 \rightarrow \text{output channel 2}. \]
\[ \gamma^3_2 \rightarrow \text{output channel 2}. \]

This triggers all packets in \( Q_{21} \) to be de-queued to output channel 2.

At the end, all packets in message 3 will have been sent to output channel 1, packets in message 2 to channel 2 followed by packets in message 1.

8. Conclusion

It is critical for developing software system to use or re-use well-defined components. Most typical data structures are common components used in software development and they are well defined. Some variations of these data structures have also been proposed in the past. One of these components is queue. In this work, we investigate an extension of the regular queue to allow the queue to have multiple output channels. Each of the output channel is the destination specified in an input item. This scenario is similar to a restricted message router with only one input channel. That is, all senders (or just one sender) send messages to the same input channel of the router, and let the router to deliver to targeted receivers (output channels).

In this paper, we specified the behavior of the queue in terms of stream functions mapping between input and output streams. The input stream contains enq and deq commands as does a regular queue. However, the packet in each enq command includes information about the message and the intended receiver, in addition to the data value. The queue needs to figure out the delivering order of the packets to satisfy the requirements of such a router. To achieve this required behavior, a re-ordering (or permutation) operation of the commands in the input stream may be necessary. We defined one such re-ordering operation.

A general implementation strategy is needed based on the specification of the behavior, such as a state transition machine. At the time of this writing, we have not developed such strategy. Instead, a particular implementation is given as an example to show how such a queue can be implemented. It uses a set of dynamically created regular queues as temporary storage to hold packets to achieve the queue permutation effect. These temporary queues act like buffers to hold packets that are not yet to be delivered according to the message routing requirements, and release them once the requirement conditions are met.

We are currently investigating models and approaches to extend the permutable queue to apply to message routers with multiple input channels as well as multiple output channels. And, we are also studying the case when the packets of the same message are out of order when entered to the queue.

References


