**Distributed Computing manuscript No.**
(will be inserted by the editor)

# On the Path from Total Order to Database Replication

**Yair Amir, Ciprian Tutu**

Johns Hopkins University
e-mail: {yairamir, ciprian}@cnds.jhu.edu

The date of receipt and acceptance will be inserted by the editor

**Summary.** We introduce ZBCast, a primitive that provides Persistent Global Total Order for messages delivered to a group of participants that can crash and subsequently recover and that can become temporarily partitioned and then remerge due to network conditions. The paper presents in detail and proves the correctness of an efficient algorithm that implements ZBCast on top of existing group communication infrastructure. The algorithm minimizes the amount of required forced disk writes and avoids the need for application level (end-to-end) acknowledgments per message. We also present an extension of the algorithm that allows dynamic addition or removal of participants.

We indicate how ZBCast can be employed to build a generic data replication engine that can be used to provide consistent synchronous database replication. We provide experimental results that indicate the efficiency of the approach.

**Key words:** Group Communication – Persistent Global Total Order – Replication – Extended Virtual Synchrony

## 1 Introduction

One of the most widely studied problems in group communication research is the problem of Atomic (Totally Ordered) Broadcast [17,13] a service that guarantees that all correct processes deliver the same set of messages in the same *agreed order*. Atomic Broadcast is used as a building block for a large number of more complex algorithms that provide distributed commit or data replication. However, Atomic Broadcast is traditionally defined under a system model that only supports node crashes, but does not allow a crashed node to recover and does not consider network partitions and remerges.

We introduce a new broadcast primitive providing *Persistent Global Total Order* (PGTO). We call this primitive *ZBcast*. ZBcast guarantees reliable delivery of messages in *agreed* order that is *global* across network partitions and merges and is *persistent*, having a lasting effect across node crashes and recoveries. We present an algorithm that relies on already defined group communication properties (Extended Virtual Synchrony, Safe Delivery) in order to provide PGTO. We formally prove the correctness of the algorithm and we extend the solution to support online additions of completely new participants and complete removals of existing members while maintaining the PGTO guarantees.

We justify the importance of such an algorithm by showing how it can be employed as a core component in order to provide efficient data replication under the strictest consistency models. Given this usage, we place additional emphasis on optimizing the algorithm performance. In our approach, we avoid the use of end-to-end acknowledgements per message. End-to-end acknowledgements are only used once for every network connectivity change event (such as network partition or merge). Furthermore, in order to provide persistent order in a system that exhibits "partial amnesia" (upon recovery from a crash all data that was not forcefully written to disk, is lost) the number of forced disk write operations is another critical performance factor. We design our algorithm such that only one forced disk operation per message is required in the system during normal operation.

The rest of the paper is organized as follows. The following subsection discusses related work. Section 2 describes the working model. Section 3 introduces a conceptual solution. Section 4 addresses the problems exhibited by the conceptual solution in a partitionable system and section 5 presents the complete ZBCast algorithm and proves its correctness. Section 6 extends the algorithm to support online removals and additions to the set of participating servers. Section 7 shows how the ZBCast primitive can be used to build a database replication scheme and discusses some of the tradeoffs of the approach, while section 8 evaluates the performance of our algorithm. We conclude in section 10.

## 2 System Model

We consider a set of nodes (servers) S=$\{S_1, S_2, ..., S_n\}$, that communicate through message passing. We refer to the nodes as servers rather than processes in order to emphasize the difference from memory-less processes used in the traditional model descriptions. Initially, we assume that the set S is fixed

and known in advance. Later, in Section 6, we will show how to deal with online changes to the set of servers[1].

### *2.1 Failure and Communication Model*

The nodes communicate by exchanging messages. The messages can be lost, servers may crash and network partitions may occur. A server that crashes may subsequently recover retaining its old identifier and stable storage. We assume no message corruption and no Byzantine faults.

The network may partition into a finite number of disconnected components. Nodes situated in different components cannot exchange messages, while those situated in the same component can continue communicating. Two or more components may subsequently merge to form a larger component.

We employ the services of a *group communication layer* which provides reliable multicast messaging with ordering guarantees (FIFO, causal, total order). The group communication system also provides a membership notification service, informing the PGTO algorithm about the nodes that can be reached in the current component. The notification occurs each time a connectivity change, a server crash or recovery, or a voluntary join/leave occurs. The set of participants that can be reached by a server at a given moment in time is called a *view* or *component*. The PGTO layer handles the server crashes and network partitions using the notifications provided by the group communication. The basic property provided by the group communication system is called Virtual Synchrony [10] and it guarantees that processes moving together from one view to another deliver the same (ordered) set of messages in the former view.

### *2.2 Service Model*

The ZBCast layer interacts externally with the application and internally with the Group Communication layer. Externally the ZBCast layer is characterized by the following signature:

input **ZBCast**($m$) - the client application submits message $m$ to be ordered in the PGTO.
output **ZDeliver**($m$) - $m$ satisfies the PGTO guarantees and is delivered to the target application[2].

Internally, the ZBCast layer interacts with the Group Communication layer according to the following signature:

input **GCDeliver**($m$,$type$) - message delivery from the group communication layer. The delivery satisfies the properties defined by the delivery $type$ (FIFO, Causal, Agreed, Safe) [3].
input **ViewChange** - A view change notification issued by the group communication as a result of a network event (partition/merge) or of server crash/recovery.
output **GCBCast**($m$,$type$) - invocation of the group communication layer broadcast primitive, specifying the desired delivery type. It is common practice to refer to the specific broadcast primitives for each delivery type instead of the generic GBCast primitive: RBCast, FIFOBCast, TOBCast, SafeBCast.

The ZDeliver primitive satisfies the following safety (correctness) properties:

**Global FIFO Order** - If server $r$ ZDelivered a message $m$ generated by server $s$, then $r$ already ZDelivered every message that $s$ generated prior to $m$.

$m^{s,i}_{r,j} \Rightarrow$for all $i$'$< i$ there exist $j$'¡$j$ such that $m^{s,i'}_{r,j'}$.

**Global Total Order** - If both servers $s$ and $r$ ZDelivered their $i$th message then these messages are identical.

$\exists m_{s,i}, m_{r,i} \Rightarrow m_{s,i} = m_{r,i}$ .

Note that the global FIFO order and the global total order invariants imply that the global total order is also consistent with causal delivery.

In order to specify the liveness properties of the PGTO algorithm, we assume two properties regarding the behaviour of the group communication layer.

1. If there exists a set of servers containing $s$ and $r$, and a time, from which on that set does not face any communication or server failure, then the group communication eventually delivers a view change containing $s$ and $r$. Moreover, we assume that if no message is lost within this set, the group communication will not deliver another view change to $s$ or $r$.
2. If a message is deliverable by the group communication then the group communication layer eventually delivers it.

We would like to note that there exist in practice a number of group communication systems (Transis [3], Spread [30], Ensemble [14], etc.) that do behave according to these assumptions.

Intuitively, the liveness property of the PGTO algorithm requires that any message that was submitted to the algorithm through the ZBCast invocation, will be ZDelivered as long as network conditions are stable and servers do not fail for a sufficiently long period of time. In order to formally specify this property we introduce the notion of *primary component* which will be presented in detail in the following sections and we rely on the concept of *causal past* as defined by Lamport [8,9].

The algorithm uniquely identifies, at any given moment during its run, at most one of the connected components to be the *primary component*. The algorithm determines, through the use of a quorum function, a unique sequence of primary components as we prove in section 5.2. According to [8,9], a message $m$ generated by server $s$ is said to be in the causal past of a server $r$ if there exists an event $e$ on $r$ and a causal dependency chain between the creation of $m$ and $e$. With these terms, we can specify the two liveness properties of our algorithm.

[1] Note that these are changes to the system setup, not view changes caused by temporary network events.

[2] The set of target applications may be different from the client applications. One such example of usage is described in Section 7

[3] The specifications of the relevant delivery types and the associated broadcast primitives will be presented, as needed, later in the paper

**Liveness** - If server $s$ receives a ZBCast($m$) invocation from a client, and there exists a primary component PC(px) which does not face any communication or server failures and a server $r$ that installed PC(px) such that $m$ is in the causal past of $r$, then $r$ orders (ZDelivers) $m$.

**ZDelivery propagation** - If server $s$ ZDelivers message $m$ and there exists a set of servers containing $s$ and $r$, and a time from which on that set does not face any communication or server failure, then server $r$ eventually ZDelivers message $m$.

$\Diamond(\exists m_{s,i} \wedge stable\_system(s,r)) \Rightarrow \Diamond \exists m_{r,i}$.

As messages are submitted to the ZBCast algorithm for ordering, they are encapsulated in a data structure which contains additional fields necessary for the establishing of the PGT Order. which we refer to as *actions*. The detailed content of an action is presented in Section 5. Throughout the paper we use predominantly the term action to refer to messages inside the ZBCast layer, as it incorporates the logic associated by the algorithm with each message and simplifies the exposition.

## 3 Total Order based Algorithm

Our solution uses a group communication layer that provides a partitionable membership service. This means that, in the presence of network partitions, the group communication layer delivers ViewChange notifications to the members of the group informing them of the current *view* which includes a list of all the nodes that are currently reachable. Nodes in different components, will receive different, disjoint views that reflect the communication barrier between the partitions. All the nodes continue executing the designated algorithm unless they crash, regardless of the view that they are part of. In order to provide a unique Global Total Order, the ZBCast layer identifies at most a single view among the existing parallel views in the server group as the *primary view* or *primary component*; the other components of a partitioned group are *non-primary* components (views). Only nodes in the primary view continue ordering actions and ZDelivering them. In essence, the ZBCast layer builds a primary-membership service on top of the partitionable membership service provided by the group communication. While defining a primary view is necessary for the correctness of the algorithm, the use of an underlying partitionable-membership service allows for significant performance optimizations enabling techniques such as eventual path propagation and tentative message pre-ordering which will be presented below.

Each server builds its own knowledge about the order of actions in the system as delivered by the employed group communication primitive. We use the coloring model defined in [1] to indicate the knowledge level associated with each action. Each server marks the actions delivered to it with one of the following colors:

Red Action An action that has been ordered within the local component by the group communication layer, but for which the server cannot, as yet, determine the global order.

Green Action An action for which the server has determined the global order.

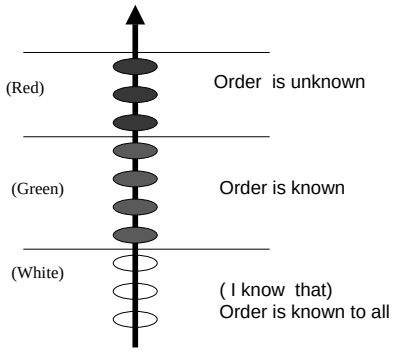


**Fig. 1.** Action coloring

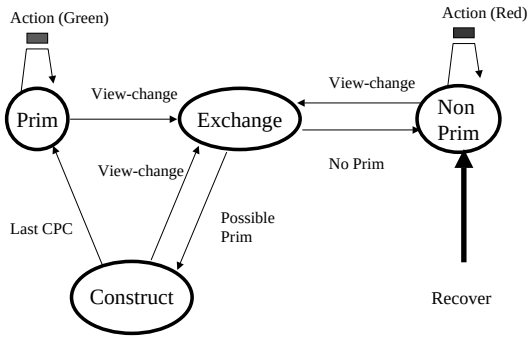


**Fig. 2.** Conceptual PGTO Algorithm

White Action An action for which the server knows that all of the servers in $S$ have already marked it as *green*. These actions can be discarded since no other server will need them subsequently.

At each server, the *white* actions precede the *green* actions which, in turn, precede the *red* ones. An action can be marked differently at different servers; however, no action can be marked *white* by one server while it is missing or is marked *red* at another server.

The actions delivered by the group communication layer to the PGTO layer in a primary component are marked green. Green actions can be ZDelivered immediately while maintaining the strictest consistency requirements. In contrast, the actions delivered in a non-primary component are marked red. The global order of these actions cannot be determined yet, so, under the strong consistency requirements, these actions cannot be ZDelivered at this stage.

### *3.1 Conceptual Algorithm*

The algorithm in this section presents a high-level solution to the PGTO problem. The algorithm is however not complete, as it is not able to deal with some of the more subtle issues that can arise in a partitionable system. We present this over-simplified solution in order to provide a better insight into some of the problems that the complete solution needs to cope with and to introduce the key properties of the final algorithm.

Figure 2 presents the state machine associated with the conceptual algorithm. A server can be in one of the following four states:

- **Prim State.** The server belongs to the primary component. When a client submits a request, it is multicast using the group communication to all the servers in the component. When a message is delivered by the group communication system to the PGTO layer, the action is immediately marked green and is ZDelivered.
- **NonPrim State.** The server belongs to a non-primary component. Client actions are ordered within the component using the group communication system. When a message containing an action is delivered by the group communication system, it is immediately marked red.
- **Exchange State.** A server switches to this state upon delivery of a view-change notification from the group communication system. All the servers in the new view will exchange information allowing them to define the set of actions that are known by some of them but not by all. These actions are subsequently exchanged and each server will ZDeliver the green actions that it gained knowledge of. After this exchange is finished each server can check whether the current view can form the next primary component. This check can be done locally, without additional exchange of messages, based on the information collected in the initial stage of the Exchange state. If the view can form the next primary component, the server will move to the Construct state, otherwise it will return to the NonPrim state.
- **Construct State.** In this state, all the servers in the component have ZDelivered the same set of actions and have the same set of red messages (they synchronized in the Exchange state) and can attempt to install the next primary component. For that, they will send a Create Primary Component (CPC) message. When a server has received CPC messages from all the members of the current component it will transform all its red messages into green, ZDeliver them in order and then switch to the Prim state. If a view change occurs before receiving all CPC messages, the server returns to the Exchange state.

In a system that is subject to partitioning we must ensure that two different components do not order actions differently thus breaking the global total order property of ZDelivery. We use a quorum mechanism to allow the selection of a unique primary component from among the disconnected components. Only the servers in the primary component will be permitted to ZDeliver actions. While several types of quorums could be used, we opted to use *dynamic linear voting* [18]. Under this system, the component that contains a (weighted) majority of the last primary component becomes the new primary component.

In many systems, processes exchange information only as long as they have a direct and continuous connection. In contrast, our algorithm propagates information by means of *eventual path*: when a new component is formed, the servers exchange knowledge regarding the actions they have, their order and color. This exchange process is only invoked immediately after a view change. Furthermore, all the components exhibit this behavior, whether they will form a primary or non-primary component. This allows the information to be disseminated even in non-primary components, reducing the amount of data exchange that needs to be performed once a server joins the primary component. In our algorithm, the eventual path translates into practice the notion of *causal past* defined in [8,9] and used in section 2.2. As such, the notion of eventual path between servers $s$ and $r$ can be viewed as a temporal communication path $\{(s,s_1), (s_1,s_2)...(s_i,r)\}$ such that any pair of servers along the path is connected by a stable network and neither server in the pair crashes for a period of time sufficient to complete the Exchange phase. Thus the information known to $s$ is gradually passed on, from server to server, until it reaches $r$. We formally define the exact information that needs to be exchanged during this phase, in section 5.

## 4 From Total Order to Persistent Global Total Order

Unfortunately, due to the asynchronous nature of the system model, we cannot reach complete common knowledge about which messages were received by which servers just before a network partition occurs or a server crashes. In fact, it has been proven that reaching consensus in asynchronous environments with the possibility of even one failure is impossible [16]. Group communication primitives based on Virtual Synchrony do not provide any guarantees of message delivery that span network partitions or server crashes/recoveries. In our algorithm, it is important to be able to tell whether a message that was delivered to one server right before a view change, was also delivered to all its intended recipients.

A server *p* cannot know, for example, whether the last actions it received from the group communication in the Prim state, before a view-change event occurred, were delivered to all the members of the primary component; Virtual Synchrony guarantees this fact only for the servers that will install the next view together with *p*. Such uncertain messages cannot be immediately marked *green* by *p*, because of the possibility that a subset of the initial membership, big enough to construct the next primary component, did not receive the messages. This subset could install the new primary component and then ZDeliver other actions, breaking consistency with the rest of the servers. This problem will manifest itself in any algorithm that tries to operate in the presence of network partitions and remerges. A solution based on Total Order and Virtual Synchrony cannot be correct in this setting without further enhancement.

### *4.1 Extended Virtual Synchrony*

In order to circumvent the inability to know who received the last messages sent before a network event occurs we use an enhanced group communication paradigm called *Extended Virtual Synchrony* (EVS) [25] that reduces the ambiguity associated with the decision problem. Instead of having to decide on two possible values, EVS creates three possible cases. To achieve this, EVS splits the view-change notification into two notifications: a *transitional* configuration (view) change message and a *regular* configuration change message. The transitional configuration message defines a reduced membership containing members of the next regular view coming directly from the same regular view. This allows the introduction of another form of message delivery, *safe delivery*, which maintains the total order property but also guarantees

that every message delivered to any process that is a member of a view is delivered to every process that is a member of that view, unless that process crashes. Messages that do not meet the requirements for safe delivery, but are received by the group communication layer, are delivered in the transitional view. No new messages are sent by the group communication in the transitional view.

The safe delivery property provides a valuable tool to deal with the incomplete knowledge in the presence of network failures or server crashes. We distinguish now three possible cases:

1. A safe message is delivered in the regular view. All guarantees are met and everyone in the view will deliver the message (either in the regular view or in the following transitional view) unless they crash.
2. A safe message is delivered in the transitional view. This message was received by the group communication layer just before a partition occurs. The group communication layer cannot tell whether other components that split from the previous component received and will deliver this message.
3. A safe message was sent just before a partition occurred, but it was not received by the group communication layer in some detached component. The message will, obviously, not be delivered at the detached component.

The power of this differentiation lies in the fact that, with respect to the same message, it is impossible for one server to be in case 1, while another is in case 3. To illustrate the use of this property consider the Construct phase of our algorithm: If a server *p* receives all CPC messages in the regular view, it knows that every server in that view will receive all the messages before the next regular view is delivered, unless they crash; some servers may, however, receive some of the CPC messages in a transitional view. Conversely, if a server *q* receives a view change for a new regular view before receiving all of the CPC messages, then no server could have received a message that *q* did not receive as safe in the previous view. In particular, no server received all of the CPC messages as safe in the previous regular view. Thus q will know that it is in case 3 and no other server is in case 1. Finally, if a server *r* received all CPC messages, but some of those were delivered in a transitional view, then *r* cannot know whether there is a server *p* that received all CPC messages in the regular view or whether there is a server *q* that did not receive some of the CPC messages at all; *r* does, however, know that there cannot exist both *p* and *q* as described.

A list of the EVS properties is presented in Appendix A.

## 5 ZBCast Algorithm

Based on the above observations the algorithm skeleton presented in Figure 2 needs to be refined. We will take advantage of the Safe delivery properties and of the differentiated view change notification that EVS provides. The two delicate states are, as mentioned, Prim and Construct.[4]

[4] While the same problem manifests itself in any state, it is only these two states where knowledge about the message delivery is critical, as it determines either the global total order (in Prim) or the creation of the new primary (Construct).

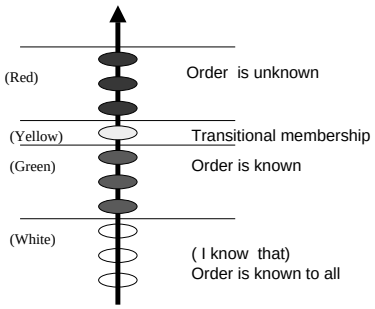


**Fig. 3.** Updated coloring model

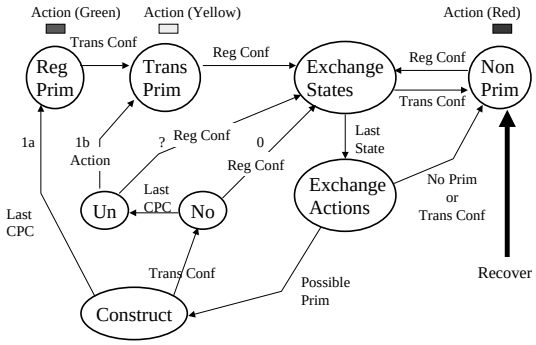


**Fig. 4.** PGTO State Machine

In the **Prim** state, only actions that are delivered as safe during the regular view can be ZDelivered. Actions that were delivered by the group communication in the transitional view cannot be marked as green and ZDelivered before we know that the next regular view will be the one defining the primary component of the system. If an action *a* that is delivered in the transitional membership is marked directly as green and ZDelivered, then it is possible that one of the detached components that did not receive this action would install the next primary component and would continue ZDelivering new actions, without ZDelivering *a*, thus breaking the consistency of the ZBcast delivery. To avoid this situation, the **Prim** state was split into two states: ***RegPrim*** and ***TransPrim*** and a new message color was introduced to the coloring model:

Yellow Action An action that was delivered in a transitional view of a primary component.

A *yellow* action becomes *green* at a server as soon as this server learns that another server marked the action *green* or when this server becomes part of a primary component. As discussed in the previous section, if an action is marked as *yellow* at some server p, then there cannot exist r and s, in this component, such that one marked the action as *red* and the other marked it *green*.

In the presence of consecutive network changes, the process of installing a new primary component can be interrupted by another view change. If a transitional view change notification is received by a server *p* while in the **Construct** state, before receiving all the CPC messages, the server will not be able to install the new primary and will switch to a new state: ***No***. In this state *p* expects to receive the delivery of the

new regular view which will trigger the initiation of a new exchange round. However, if *p* receives all the rest of the CPC messages in **No** (i.e. in the transitional view), it means that it is possible that some server *q* has received all the CPC messages in **Construct** and has moved to **RegPrim**, completing the installation of the new primary.

To account for this possibility, *p* will switch to another new state: ***Un*** (undecided). If an action message is received in this state then *p* will know for sure that there was a server *q* that switched to **RegPrim** and even managed to generate new actions before noticing the network failure that caused the cascaded membership change. Server *p,* in this situation (1b), has to act as if installing the primary component in order to be consistent, mark its old yellow/red actions as green, mark the received action as yellow and switch to **TransPrim**, "joining" *q* who will come from **RegPrim** as it will also eventually notice the new view change. If the regular ViewChange message is delivered without any message being received in the **Un** state (transition marked ? in Figure 4), *p* remains uncertain whether there was a server that installed the primary component and will not attempt to participate in the formation of a new primary until this dilemma is cleared through exchange of information with one or, in the worst case, all of the members that tried to install the same primary as *p*.

Figure 4 shows the updated state machine. Aside from the changes already mentioned, the Exchange state was also split into **ExchangeStates** and **ExchangeActions**, mainly for clarity reasons. From a procedural point of view, once a view change is delivered, the members of each view will try to establish a maximal common state that can be reached by combining the information and actions held by each server. After the common state is determined, the participants proceed to exchange the relevant actions. Obviously, if the new membership is a subset of the old one, there is no need for action exchange, as the states are already synchronized.

### *5.1 Algorithm Pseudocode*

In this subsection we present the complete pseudocode of the algorithm. The proof of correctness in subsection 5.2 will be based on the code presented in this section. We start the presentation by describing the structure of the data and of the messages used by the algorithm.

***Data Structure***

The structure *Action_id* contains two fields: *server_id* the creating server identifier, and *action_index*, the index of the action created at that server.

The following local variables reside at each of the servers. Additional variables may appear in the pseudocode, internal to various procedures.

- *serverId* - a unique identifier of this server in the servers group.
- *actionIndex* - the index of the next action created at this server. Each created action is stamped with the *actionIndex* after it is incremented.
- *conf* - the current configuration (view) of servers delivered by the group communication layer. Contains the following fields:
  *conf_id* - identifier of the configuration.
  *set* - the membership of the current connected servers.
- *attemptIndex* - the index of the last attempt to form a primary component.
- *primComponent* - the last primary component known to this server. It contains the following fields:
  *prim_index* - the index of the last primary component installed.
  *attempt_index*- the index of attempt by which the last primary component was installed.
  *servers* - identifiers of participating servers in the last primary component.
- *state* - the state of the algorithm. One of {RegPrim, TransPrim, ExchangeStates, ExchangeActions, Construct, No, Un, NonPrim}.
- *actionQueue* - ordered list of all the red, yellow and green actions. White actions can be discarded and therefore, in a practical implementation, are not in the *actionQueue*. For the sake of easy proofs the presented algorithm does not extract actions from the *actionQueue*. Refer to [1] for details concerning message discarding.
- *ongoingQueue* - list of actions generated at the local server. Such actions that were delivered and written to disk can be discarded. This queue protects the server from losing its own actions due to crashes (power failures).
- *redCut* - array[1..n] - $redcut[i]$ is the index of the last action server $i$ has sent and that this server has.
- *greenLines* - array[1..n] - $greenLines[i]$ is the identifier of the last action server $i$ has marked green as far as this server knows. *greenLines*[*serverId*] represents this server's green line.
- *stateMessages* - a list of State messages delivered for this view.
- *vulnerable* - a record used to determine the status of the last installation attempt known to this server. It contains the following fields:
  *status* - one of {Invalid, Valid}.
  *prim_index* - index of the last primary component installed before this attempt was made.
  *attemp_index* - index of this attempt to install a new primary component.
  *set* - array of *serverId*s trying to install this new primary component.
  *bits* - array of bits, each of {Unset, Set}.
- *yellow* - a record used to determine the yellow actions set. It contains the following fields:
  *status* - one of {Invalid, Valid}
  *set* - an ordered set of action identifiers that are marked yellow.

***Message Structure***

Three types of messages are created by the ZBCast layer:

- **Action_message** - a regular action message contains the following fields:
  *type* - the message type (Action)
  *action_id* - Action_id type datastructure identifying this action.
  *green_line* - the identifier of the last action marked green at the creating server at the time of creation.
  *client* - the identifier of the client requesting this action.
  *data* - the actual action data.
- **State_message** - contains the following fields:
  *type* - the message type (State)

*server_id*, *green_line*, *attempt_index*, *prim_component*, *conf_id*, *red_cut*, *vulnerable*, *yellow* - the corresponding data structures at the creating server.

- **CPC_message** - contains the following fields:
  *type* - the message type (CPC)
  *server_id*, *conf_id* - the corresponding data structures at the creating server.

***Definition of Events***

The events handled by the ZBCast layer follow the signature described in section 2.2 and correspond to interactions with either the GC layer or the client application. In order to simplify the presentation in the following sections we will shorthand the notation for GC delivery of different message types by using the respective message types as event names. Similarly, we will refer to the delivery of view change notifications by the type of view change event. This conventions lead to the following event notation:

- **Action** - an action message was delivered by the group communication layer (GCDeliver(action_mess)).
- **State_mess** - a state message was delivered by the group communication layer (GCDeliver(state_mess)).
- **CPC_mess** - a Create Primary Component message was delivered by the group communication layer (GCDeliver(CPC_mess)).
- **Reg_conf** - a regular configuration was delivered by the group communication layer (ViewChange(conf, regular)).
- **Trans_conf** - a transitional configuration was delivered by the group communication layer (ViewChange(conf, trans)).
- **Client_req** - a client request was received from a client that issued ZBCast(client_mess).

**CodeSegment 5.1** Code executed in the NonPrim State

```
case event is
Action:  MarkRed( Action )
Reg_conf:
    set conf according to Reg_conf
    Shift_to_exchange_states()
Trans_conf, State_mess:  Ignore
Client_req:  actionIndex++
    create action_mess and write to ongoingQueue
    ** sync to disk
    SafeBCast(action_mess)
CPC_mess:  Not possible
```

**CodeSegment 5.2** Code executed in the RegPrim State

```
case  event is
Action:  MarkGreen( Action )    ( OR-1.1 )
    greenLines[ Action.action_id.server_id ] = Action.green_line
Trans_conf:  State = TransPrim
Client_req:  actionIndex++
    create action_mess and write to ongoingQueue
    ** sync to disk
    SafeBCast(action_mess)
Reg_conf, State_mess, CPC_mess:  Not possible
```

**CodeSegment 5.3** Code executed in the TransPrim State

```
case  event is
Action:  MarkYellow( Action )
Reg_conf:  set Conf according to Reg_conf
    vulnerable.status = Invalid
    yellow.status = Valid
    Shift_to_exchange_states()
Client_req:  buffer request
Trans_conf, State_mess, CPC_mess:  Not possible
```

**CodeSegment 5.4** Code executed in the ExchangeStates state

```
case  event is
Trans_conf:  state = NonPrim
State_mess:
    If (State_mess.conf_id = Conf.conf_id )
        add State_mess to stateMessages
    if ( all state messages were delivered )
        if ( most updated server ) Retrans()
        Shift_to_Exchange_actions()
Action:  MarkRed( Action )
CPC_mess:  Ignore
Client_req:  buffer request
Reg_conf:  Not possible
```

### *5.2 Proof of Correctness*

In this section we prove that the ZBCast protocol maintains the safety and liveness criteria defined in Section 2.2. We assume that the group communication layer maintains extended virtual synchrony.

***Notations***

- An action $a$ is *ordered* by server $s$ when the ZDeliver procedure is invoked for $a$ at $s$[5]. We say server $s$ has action $a$ when the ApplyRed procedure is invoked for $a$ at $s$.
- $S$ - all of the members of the *servers group*.
- $a_{r,j}^{s,i}$ −action $a$ is the $i$th action generated (SafeBCasted) by server $s$, and the $j$th action ordered (ZDelivered) by server $r$. Notations such as $a_{r,j}$ and $a^{s,i}$ are also possible where the generating/ordering server is not important.
- We use the term *send* to refer to the algorithm invocation of a GCBCast primitive, while the terms *receive* and *deliver* refer to the GDelivery of a message by the group communication layer or the detection of a ViewChange notification.
- The pair (*px*,*ax*) represents the *prim_index* and *attempt_index* of the *primComponent* structure. We say that (*px*,*ax*) $>$ (*px*',*ax*') iff either *px* $>$ *px*' or *px* = *px*' and *ax* $>$ *ax*'.
- $PC_s(px, ax)$−server $s$ installed or learned about primary component with *px* as primary_index and with *ax* as attempt index. Notations such as $PC_s(px)$, and $PC(px)$are

[5] In practice, server $s$ may determine the PGT order of an action $a$ but cannot be certain of its ZDelivery to the aplication in case it crashes at a critical moment. The only way for the algorithm to guarantee exactly-once delivery is to query the application and confirm the delivery of the action. The algorithm can however guarantee exactly-once ordering of the action which is why we will predominantly use this term in this section.

**CodeSegment 5.5** Code for the *Shift_to_exchange_states*, *Shift_to_exchange_actions*, and *End_of_retrans Procedures*

***Shift_to_exchange_states()***
** sync to disk
clear stateMessages
SafeBCast(State_mess)
*state* = ExchangeStates
***Shift_to_exchange_actions()***
*state* = ExchangeActions
if ( end of retransmission ) End_of_retrans()
***End_of_retrans()***
Incorporate all green_line from State messages to *greenLines*
ComputeKnowledge()
if ( IsQuorum() )
  *attemptIndex*++
  *vulnerable.status* = Valid
  *vulnerable.prim_index* = *primComponent.prim_index*
  *vulnerable.attempt_index* = *attemptIndex*
  *vulnerable.set* = *conf.set*
  *vulnerable.bits* = all Unset
  ** sync to disk
  SafeBCast(CPC_mess)
  *state* = Construct
else
  ** sync to disk
  Handle_buff_requests()
  *state* = NonPrim

**CodeSegment 5.6** Code executed in the ExchangeActions State

case event is
Action:
  Mark Action according to State messages ( OR-3 )
  if ( turn to retransmit ) Retrans()
  if ( end of retransmission ) End_of_retrans()
Trans_conf: *state* = NonPrim
Client_req: buffer request
Reg_conf, State_mess, CPC_mess: Not possible

also possible when the missing parameters are not important.

- We say that server $s$ is a member of *PC*(*px*) if $s$ is in the servers set of *PC*(*px*).

### 5.2.1 Safety

We prove that the following properties are invariants of the protocol. i.e. they are maintained throughout the execution of the protocol:

**Global FIFO Order** - If server $r$ ZDelivered an action $a$ generated by server $s$, then $r$ already performed every action that $s$ generated prior to $a$.

$a_{r,j}^{s,i} \Rightarrow$for all $i$'$< i$ there exist $j$'¡$j$ such that $a_{r,j'}^{s,i'}$.

**Global Total Order** - If both servers $s$ and $r$ ZDelivered their $i$th actions then these actions are identical.

$\exists a_{s,i}, a_{r,i} \Rightarrow a_{s,i} = a_{r,i}$.

**CodeSegment 5.7** ComputeKnowledge

1. *primComponent* = *primComponent* in all State messages with the maximal (primIndex, attemptIndex)
   *updatedGroup* = the servers that sent *primComponent* in their State message
   *validGroup* = the servers in *updatedGroup* that sent Valid *yellow.status*
   *attemptIndex* = max *attemptIndex* sent by a server in *updatedGroup* in their State message

2. if *validGroup* is not empty
   *yellow.status* = Valid
   *yellow.set* = intersection of *yellow.set* sent by *validGroup*
  else
   *yellow.status* = Invalid

3. for each server with Valid in *vulnerable.status*
   if ( *serverId* not in *primComponent.set* or one of its *vulnerable.set* does not have identical *vulnerable.status* or *vulnerable.prim_index* or *vulnerable.attempt_index* ) then Invalid its *vulnerable.status*
4. for each server with Valid in *vulnerable.status*
   set its *vulnerable.bits* to union of *vulnerable.bits* of all servers with Valid in *vulnerable.status*
   if all bits in its *vulnerable.bits* are set then its *vulnerable.status* = Invalid

**CodeSegment 5.8** Code of the IsQuorum and Handle_buff_requests Procedures

**IsQuorum()**
  if there exists a server in *conf* with *vulnerable.status* = Valid return False
  if *conf* does not contain a majority of *primComponent.set* return False
  return True

**Handle_buff_requests**()
  for all buffered requests
    *actionIndex*++
    create action_mess and write to *ongoingQueue*
  ** sync to disk
  for all buffered requests
    SafeBCast(action_mess)
  clear buffered requests

**CodeSegment 5.9** Code executed in the Construct state

case event is
Trans_conf: *state* = No
CPC_mess:
  if ( all CPC messages were delivered )
    for each server s in *conf.set*
      set *greenLines*[s] to *greenLines*[*serverId*]
    Install()
    *state* = RegPrim
    Handle_buff_requests()
Client_req: buffer request
Action, Reg_conf, State_mess: Not possible

**CodeSegment 5.10** Install procedure

```
if ( yellow.status = Valid )
    for all actions in yellow.set
        MarkGreen(Action)          ( OR-1.2 )
yellow.status = Invalid
yellow.set = empty
primComponent.prim_index++
primComponent.attempt_index = attemptIndex
primComponent.servers = vulnerable.set
attemptIndex = 0
for all red actions ordered by Action.action_id
    MarkGreen(Action)              ( OR-2 )
** sync to disk
```

**CodeSegment 5.11** Code executed in the No state

```
case event is
Reg_conf:
    set conf according to Reg_conf
    vulnerable.status = Invalid
    Shift_to_exchange_states()
CPC_mess: if ( all CPC messages were delivered ) state = Un
Client_req: buffer request
Action, Trans_conf, State_mess: Not_possible
```

**CodeSegment 5.12** Code executed in the Un state

```
case event is
Reg_conf:
    set conf according to Reg_conf
    Shift_to_exchange_states()
Action:
    Install()
    MarkYellow( Action )
    state = TransPrim
Client_req: buffer request
Trans_conf, State_mess, CPC_mess: Not possible
```

**CodeSegment 5.13** Recover procedure

```
state = NonPrim
for each action in ongoingQueue
  if ( redCut[ serverId ] < Action.action_id.action_index)
    MarkRed( Action )
** sync to disk
```

Note that the global FIFO order and the global total order invariants imply that the global total order is also consistent with causal delivery.

We assume that all of the servers start with the following initial state: *primComponent.prim_index*=0, *primComponent.attempt_index=0, primComponent.servers* $= S$, empty *actionQueue, vulnerable.status*= Invalid, *yellow.status*= Invalid.

Before proving the invariants, we will prove a few claims regarding the history of primary components in the system.

**Lemma 1.** *If server r learns about* $PC_r(px, ax)$*, then there is a server s that installed* $PC_s(px, ax)$ *such that* $PC_r(px, ax) = PC_s(px, ax)$.

**CodeSegment 5.14** Marking procedures

**MarkRed( Action )**

```
if ( redCut[ Action.action_id.server_id ] = Action.action_id.action_index - 1 )
  redCut[ Action.action_id.server_id ]++
  Insert Action at top of actionQueue
  if ( Action.type = Action ) ApplyRed( Action )
    if ( Action.action_id.server_id = serverId ) delete action from ongoingQueue
```

**MarkYellow( Action )**

```
MarkRed( Action )
yellow.set = yellow.set + Action
```

**MarkGreen( Action )**

```
MarkRed( Action )
if ( Action not green )
  place action just on top of the last green action
  greenLines[ serverId ] = Action.action_id
  ZDeliver( Action )
```

*Proof.* A server $r$ knows about *PC*(*px, ax*) either when installing it or when learning about it. From the algorithm, the only place $r$ learns about $PC_r(px, ax)$ is at Step 1 of the Compute_knowledge procedure (CodeSegment 5.7). According to Step 1, there is a server $t$ that sent a State message containing $PC_r(px, ax)$. Therefore, to start the chain, there must be a server $s$ that installed $PC_s(px, ax)$ such that $PC_r(px, ax) = PC_s(px, ax)$. □

**Lemma 2.** *The pair (px,ax) never decreases at any server s; Moreover, it increases each time server s sends a CPC message or installs a new primary component.*

*Proof.* Before a server installs a primary component, it sends a State message containing its last known primary component (field *primComponent* in the State message). Note that just before sending the State message, the server forces its data structure to disk, so that this information is not lost if the server subsequently crashes. In the Compute_knowledge procedure (CodeSegment 5.7), the server sets *primComponent* to the maximal (*px, ax*) that was sent in one of the State messages (including its own). Therefore, the local value of (*px,ax*) does not decrease.

Just before sending the CPC message, while the server is in the Exchange_actions State, it increments *primComponent.attempt_index* and immediately forces its data structure to disk (CodeSegment 5.5).

When installing, the server increments *primComponent.prim_index* (CodeSegment 5.10) and forces its data structure to disk. Since these three places are the only places where *primComponent* may change, the claim holds. □

**Lemma 3.** *If server s installs* $PC_s(px, ax)$*, then there exists server r that installed* $PC_r(px - 1, ax')$.

*Proof.* According to the algorithm, if server $s$ installs a primary component with *primComponent.prim_index = px*, then

there is a server $t$ that sent a State message containing *primComponent.prim_index* = *px*-1 for that installation. Therefore $t$ either installed a primary component with *primComponent.prim_index* = *px*-1 or learned about it. In any case, according to Lemma 1, there exists a server $r$ that installed such primary component. □

**Lemma 4.** *If server s installed* $PC_s(px, ax)$ *and server r installed* $PC_r(px, ax')$*, then* $ax = ax'$ *and* $PC_s(px, ax) = PC_r(px, ax)$.

*Proof.* We prove this claim by induction on the primary component index *px*.

First we show that the claim holds for *px*=1:

Assume the contrary. Without loss of generality, suppose that *ax* > *ax*'. Remember that at initialization, the set of servers in *primComponent* is $S$. Therefore, since $s$ installs a primary component with (1,*ax*) there is a majority of $S$ that participated in that attempt and sent a CPC message with (0,*ax*).

For the same reason, there is a majority of $S$ that sent a CPC message with (0,*ax*'). Hence, there must be a server $t$ that participated in both attempts and sent both messages. From Lemma 2 and from the fact that *ax* > *ax*', $t$ sent the CPC message with (0,*ax*') before sending that with (0,*ax*).

From the algorithm, since $r$ installed, there is a server that received all the CPC messages of the first attempt in the regular view. i.e. there is a server belonging to the first majority, that shifted from Construct to RegPrim (CodeSegment 5.9). The safe delivery property of extended virtual synchrony (Property 12 in the Appendix) ensures that all the members of the first majority (including $t$) received all the CPC messages before the next regular view, or crashed. Therefore, according to the algorithm, for each server $u$ belonging to the first majority, only the following cases are possible:

Server $u$ receives all the CPC messages in the regular view and installs a primary component *PC*(1,*ax*') (CodeSegment 5.9).

Server $u$ crashes before processing the next regular view and remains vulnerable.

Server $u$ receives all the CPC messages, but some are delivered in the transitional view. In this case $u$ is in the Un state. Two sub-cases are possible:

Server $u$ receives an action, and installs *PC*(1,*ax*') (see CodeSegment 5.12).

Server $u$ receives the next regular view and remains vulnerable.

Since these are the only possible cases, every member of the first majority either installs *PC*(1,*ax*') or remains vulnerable. If server $t$ installs PC(1,*ax*'), then Lemma 2 contradicts the fact that it later sent a CPC message with (0,*ax*). If server $t$ remains vulnerable, then according to the algorithm, it must invalidate its vulnerability before sending another CPC message. This can happen in the only following ways:

Server $t$ learns about a higher primary component. Again, Lemma 2 contradicts the fact that it later sent a CPC message with (0,*ax*).

Server $t$ learns that all the servers from the first majority did not install and are vulnerable to the same server set, contradicting the fact that server $r$ installed $PC_r(px, ax')$.

Server $t$ learns that another server from the set is not vulnerable with the same (0,*ax*'). The only servers that are not vulnerable in this set, are the servers that installed. Hence server $t$ learns about the installation of a higher primary component before sending its second CPC message, which contradicts Lemma 2.

Therefore, no such server $t$ exists, proving the base of the induction.

The induction step assumes that the claim holds for *px* and shows that it holds for *px*+1.

The proof is exactly the same proof as for *px*=1, where 0 is replaced by *px*, 1 is replaced by *px*+1, and $S$ is replaced by the set of servers in *PC*(*px*,*ax*). □

**Lemma 5.** *If server s installed or learned about* $PC_s(px)$ *and server r installed or learned about* $PC_r(px)$*, then* $PC_s(px) = PC_r(px)$.

*Proof.* Follows directly from Lemma 1 and Lemma 4. □

From here on, we will denote *PC*(*px*) the primary component that was installed with *prim_index px*. Lemma 5 proved the uniqueness of *PC*(*px*) for all *px*.

**Lemma 6.** *If a primary component PC(px+1) is installed, then a primary component PC(px) was already installed, and there exists a server s such that s is a member of both sets of PC(px) and PC(px+1).*

*Proof.* If a primary component *PC*(*px*+1) is installed then there exists a server that installed it. According to Lemma 3, there is a server that installed *PC*(*px*). According to Lemma 5, all the servers that installed or learned about a primary component with index *px*, installed or learned about the same *PC*(*px*). According to the algorithm, a majority from *PC*(*px*) is needed to propose *PC*(*px*+1) in order to install *PC*(*px+1*), therefore there is a server that are members of both sets of *PC*(*px*) and *PC*(*px*+1). □

We are now ready to prove the invariants. There are three places where the MarkGreen procedure is called. We label them with the following labels that appear also in the protocol pseudo-code:

*OR-1* - the action was SafeBCast (sent) and GCDelivered (delivered) at a primary component. This notation covers the following two particular situations marked in the algorithm code:

*OR-1.1* - the action was delivered in the regular view of a primary component (see CodeSegment 5.2).

*OR-1.2* - the action was delivered in the transitional view of the primary component, for each member of the last primary component that participates in the quorum that installs the next primary component (see CodeSegment 5.10).

*OR-2* - The action was delivered by the GC at a non-primary component and was ordered with the installation of a new primary component (see CodeSegment 5.10).

*OR-3* - The action was ordered when this server learned about this order from another server that already ordered it (see CodeSegment 5.6).

**Lemma 7.** *If server s orders an action according to OR-3 then, there exists server r that order this action at the same order according to OR-1 or OR-2.*

*Proof.* At initialization, no actions are ordered at any of the servers in $S$, since *actionQueue* is empty. *OR-1*, *OR-2* and *OR-3* are the only possible places where actions are ordered by the algorithm. If server $s$ orders action $a$ according to *OR-3* then there was a server that SafeBCast the action $a$ and its order at the ExchangeActions state (see CodeSegments 5.4 and 5.6). To start this chain, there must be a server that ordered action $a$ in a different way, and *OR-1* and *OR-2* are the only possibilities. □

**Lemma 8.** *Assume that: servers s and r install PC(px), they have the same set of actions, ordered in the same order, and marked in the same color, in their actionQueue when sending the CPC message for PC(px). Then, for every action a that both r and s ordered in PC(px) according to OR-1.1, they ordered it at the same order.*

*Proof.* Under the assumption, and according to the algorithm $s$ and $r$ have identical *actionQueue*, *greenLines* and invalid *Yellow* when they complete the Install procedure.

Since $a$ was ordered both at $r$ and $s$ according to *OR-1.1* it was delivered both to $r$ and $s$ in the regular view $v$ within which *PC*(*px*) existed. According to the agreed delivery property of extended virtual synchrony (Property 11 in the Appendix), the same set of actions at the same order is delivered up to and including action $a$ to both $r$ and $s$. According to the algorithm, each delivered action in the RegPrim state is marked green immediately when it is delivered. Therefore, both $r$ and $s$ ordered the action $a$, and all previous actions, at the same order. □

**Lemma 9.** *Assume that: servers s and r are members of PC(px), they have the same set of actions, in the same order, and marked in the same color in their actionQueue, when sending the CPC message for PC(px), and s is a member of PC(px+1). Then, for every action a that r ordered in PC(px) according to OR-1.1, s either ordered a or has a in its Yellow at the same order before sending the CPC message for PC(px+1). Moreover, if s installs PC(px+1) then s orders a at the same order as r.*

*Proof.* Under the assumption, and according to the algorithm $s$ and $r$ have identical *actionQueue*, *greenLines* and invalid *Yellow* when they complete the Install procedure.

Since $a$ was ordered at $r$ according to *OR-1.1* it was delivered to $r$ in the regular view $v$ within which *PC*(*px*) existed. According to the safe delivery property of extended virtual synchrony (Property 12 in the Appendix), $a$ was delivered in $v$ or in *trans*($v$) to every server $t$ in *PC*(*px*) unless it crashes. According to the agreed delivery property (Property 11 in the Appendix), $a$ and all prior messages in view *com*($v$) were delivered to $t$ in the same order.

Therefore, only three cases are possible for any server $t$ in *PC*(*px*):

*Case-1* - Action $a$, and all previous actions delivered to $r$ in $v$, are delivered to $t$ in the regular view $v$. According to the algorithm, $t$ marks $a$ and all previous actions as green at the same order (see CodeSegment 5.2) according to *OR-1.1*.

*Case-2* - Action $a$, and all previous actions delivered to $r$ in $v$, are delivered to $t$ in the regular view $v$ or transitional view *trans*($v$) in the same order, and $a$ is delivered in the transitional view *trans*($v$), and the next regular view is delivered and processed at $t$. In this case, according to the algorithm, $a$ and all prior actions that are not yet ordered will be included in the *Yellow* set at the same order, and *Yellow* will be valid.

*Case-3* - Server $t$ crashed before action $a$ was processed at $t$ and before the next regular view was processed at $t$. According to the algorithm, server $t$ remains *vulnerable* after it recovers.

Consider server $s$. If Case-1 exists for $s$, then $s$ already ordered $a$ at the same order as $r$ before sending the CPC message for *PC*(*px*+1).

If Case-2 exists for $s$, then action $a$ and all previous actions in *Yellow* are not extracted from the *Yellow* of $s$ unless $s$ learns about the order of it before sending the CPC message for *PC*(*px*+1). In this case, according to Lemma 7, there is a server $u$ that already ordered this message according to *OR-1* or *OR-2*. The only possibility is that $u$ is a Case-1 server so $a$ and all previous actions were ordered at $u$ at the same order as $r$. According to the algorithm, $s$ ordered $a$ and all previous actions at the same order as $r$ at the ExchangeActions state (see CodeSegment 5.6).

If Case-3 exists for $s$ then, according to the algorithm (see CodeSegment 5.8) $s$ has to invalidate its vulnerability before it is able to send a CPC message to install *PC*(*px*+1). There are only two possibilities for $s$ to invalidate its vulnerability. The first possibility is to learn that another server that belongs to *PC*(*px*) is unvulnerable. The only unvulnerable servers are Case-1 and Case-2 servers which are more updated than $s$. If $s$ learns about them and their order (see CodeSegment 5.4 and 5.6), then the claim holds. The second possibility occurs if $s$ learns that all of the servers that belong to *PC*(*px*) are vulnerable. In this case, according to the algorithm, they all crashed before processing the next regular view. Since there exists at least one Case-1 server ($r$), the most updated server in PC(*px*) is a Case-1 server. Therefore, according to the algorithm, when $s$ learns that $r$ is also vulnerable, $s$ also learns the order of $a$ and all the previous actions at $r$.

Lastly, if $s$ installs *PC*(x+1), it marks all the yellow actions as green. □

**Lemma 10.** *If server s is a member of PC(px) then:*

*(i) s has marked as green or yellow at the same order, every action that any server marked as green according to OR-1 or OR-2 in PC(px-1).*

*(ii) s has marked as green at the same order, every action that any server marked as green according to OR-1 or OR-2 in PC(px') where px' < px-1.*

*(iii) if r is another member of PC(px), then r and s have the same set of actions, ordered in the same order, and marked in the same color, in their actionQueue, when sending the CPC message for PC(px).*

*Proof.* We prove this claim by induction on the primary component index *px*.

First we show that the claim holds for *px*=1.

At initialization, no actions are ordered at any of the servers in $S$ and *Yellow* is empty, proving ($i$) and (*ii*) for *px*=1. According to the algorithm, all the members of *PC*(1) exchange actions so that they have identical set of actions in their *actionQueue* before sending the CPC message, and all actions are red. Therefore, the claim holds for *px*=1.

The induction step assumes that the claim holds for all primary components up to and including *px* and shows that the claim holds for *px*+1.

According to the algorithm and to the delivery integrity property of extended virtual synchrony (Property 4 in the Appendix), only members of *PC*(*px*) can order actions according to OR-1 or OR-2 in PC(*px*') for any *px*'.

According to Lemma 6, there is one server $t$ that is a member of both *PC*(*px*) and *PC*(*px*+1). Let $s$ be any member of *PC*(*px*+1). Only the following two cases are possible:

*Server $s$ was a member of PC(px).* According to the induction assumption (*iii*) $s$ and $t$ had the same set of actions in the same order, and marked in the same colors when sending the CPC message for *PC*(*px*). According to Lemma 9, and since $s$ and $t$ are members of *PC*(*px*), they both had marked as green or yellow, at the same order, all actions that **any** server marked as green according to *OR-1* or *OR-2* in *PC*(*px*) are marked as green. Thus, $(i)$ is proved. Moreover, according to the induction assumption (*ii*) and Lemma 9, and the fact that there is a server that installed *PC*(*px*), they both marked as green, at the same order, all actions that **any** server marked as green according to *OR-1* or *OR-2* in *PC*(*px*') for any *px*' $<$ *px*. Thus, (*ii*) is proved. Finally, according to the algorithm, and since $r$ and $s$ are both members of *PC*(*px*+1), they both sent a CPC message for *PC*(*px*+1), which required them to go through a retransmission phase in the Exchange_actions state (see CodeSegment 5.6). Therefore, they have the same actions, ordered at the same order and marked at the same colors before sending the CPC message for *PC*(*px*+1). Thus, (*iii*) is proved.

*Server $s$ was **not** a member of PC(px).* According to the induction assumption (*ii*) $t$ has marked as green at the same order all actions that $s$ marked green according to *OR-1* and *OR-2*, proving $(i)$ and (*ii*). According to Lemma 7, any action ordered by $s$ according to *OR-3* was already ordered at the same order by another server according to *OR-1* or *OR-2*. According to the induction assumption $(i)$ and (*ii*), $t$ has marked as green or yellow at the same order as any other server that ordered it as green. Hence, $t$'s order can not contradict $s$'s order. Since they exchange actions before sending the CPC they will have the same set of red, yellow and green before sending their CPC messages. Thus, (*iii*) is also proved. □

**Theorem 1 (Global Total Order).** *If both servers s and r ZDelivered their ith actions then these actions are identical.*

$\exists a_{s,i}, a_{r,i} \Rightarrow a_{s,i} = a_{r,i}$.

*Proof.* From Lemma 10 and Lemma 8, all the servers that order an action according to *OR-1* or *OR-2* do so in the same order. From Lemma 7, if a server orders an action according to *OR-3* there already exists a server that ordered that action according to *OR-1* or *OR-2* at the same order. Therefore, since *OR-1*, *OR-2*, and *OR-3* are the only possibilities to order action, if two servers ordered an action $a$ they ordered $a$ at the same order. □

**Lemma 11.** *If r has $a^{s,i}$ such that $a^{s,i}$ was SafeBCast by s at view v and was delivered to r in v or in the transitional view immediately following v, then r already has $a^{s,j}$ for any $j < i$.*

*Proof.* According to extended virtual synchrony, both $s$ and $r$ are members of $v$.

According to the algorithm, when a regular view is delivered, then the servers exchange actions that are missed by any of them (see CodeSegment 5.4 and 5.6). If server $s$ generated $a^{s,j}$ before this retransmission then, if $r$ does not have $a^{s,j}$, it will be retransmitted. If server $s$ generated both $a^{s,j}$ and $a^{s,i}$ after the retransmission, then according to the algorithm, $a^{s,j}$ was generated first. According to the causal delivery property of extended virtual synchrony (Property 10 in the Appendix), if $a^{s,i}$ is delivered to $r$ in $v$ or the transitional view that follows $v$, then $a^{s,j}$ is delivered to $r$ before $a^{s,i}$. □

**Theorem 2 (Global FIFO Order).** *If server r ZDelivered an action a generated by server s, then r already ZDelivered every action that s generated prior to a.*

$a^{s,i}_{r,j} \Rightarrow$*for all* $i' < i$ *there exist* $j' < j$ *such that* $a^{s,i'}_{r,j'}$.

*Proof.* According to the algorithm, $s$ creates its own actions according to a FIFO order. Moreover, $s$ never looses its own actions even if it crashes (see CodeSegments 5.1, 5.2 and 5.13).

Assume the contrary. Without loss of generality, assume that $t$ is the first server that orders the $i$th action of some server $s$, $a^{s,i}$, such that the $j$th action of $s$, $a^{s,j}$, is not ordered at $t$ for some $j < i$. Therefore, according to Theorem 1, any server that orders $a^{s,i}$, orders it before ordering $a^{s,j}$.

Since $t$ is the first server to order $a^{s,i}$, only three cases are possible for $t$:

Server $t$ orders $a^{s,i}$ according to *OR-1.1*. In this case, according to the algorithm, $a^{s,i}$ is delivered in a primary component *PC*(*px*) such that $t$ is a member of *PC*(*px*). According to the delivery integrity property of extended virtual synchrony, $s$ is also a member of the regular view within which *PC*(*px*) is installed. Therefore, according to the algorithm, $s$ is also a member of *PC*(*px*). Therefore, according to Lemma 11, $a^{s,i}$ was delivered (and therefore, ordered) first. i.e. this case is not possible.

Server $t$ orders $a^{s,i}$ according to *OR-1.2*. In this case, according to the algorithm, there was a $u$ server at which $a^{s,j}$ was delivered in a transitional view of some primary component *PC*(*px*). According to the delivery integrity property of extended virtual synchrony (Property 4 in the Appendix), and to the algorithm, $s$ is a member of *PC*(*px*). If $a^{s,j}$ was generated before the installation of *PC*(*px*), then $u$ ordered it at the installation of *PC*(*px*), and before ordering $a^{s,i}$. If $a^{s,j}$ was generated after the installation of PC(px) then both $a^{s,i}$ and $a^{s,j}$ where delivered in the same view. According to the causal delivery property of extended virtual synchrony (Property 10 in the Appendix), $a^{s,i}$ was delivered (and therefore, ordered) first. i.e. this case is not possible.

Server $t$ orders $a^{s,i}$ according to *OR-2*. In this case, according to the algorithm, $t$ orders all its unordered actions according to their *Action_id*. Since $j < i$, and both $a^{s,i}$ and $a^{s,j}$ are generated by the same server, if $t$ had $a^{s,j}$ then it would have ordered it before $a^{s,i}$. Therefore, $t$ does not have $a^{s,j}$.

Therefore, since only Case-3 might be possible, there has to be a server that receives $a^{s,i}$ **before** it received $a^{s,j}$ and **before** $a^{s,i}$ is ordered by any server. Assume that $r$ is the first server to receive $a^{s,i}$ before it received $a^{s,j}$.

According to Lemma 11, server $r$ could not have $a^{s,i}$ as a new action generated by $s$ without having all prior actions generated by $s$. Therefore, according to the algorithm,

the only option left for $r$ is to have $a^{s,i}$ as a result of a retransmission. Since $a^{s,i}$ is not yet ordered, and each server that has $a^{s,i}$ has also $a^{s,j}$, and since retransmission for unordered messages is done in FIFO order, $a^{s,j}$ is retransmitted first. By the causal delivery property of extended virtual synchrony (Property 10 in the Appendix), $a^{s,j}$ is delivered to $r$ before $a^{s,i}$, leading to a contradiction. □

### 5.2.2 Liveness

In section 2.2 we specified the assumptions we make regarding the group communication layer properties. We now proceed to prove the Liveness properties hold for our algorithm.

**Theorem 3 (ZDelivery Propagation).** *If server $s$ ZDelivers action $a$ and there exists a set of servers containing $s$ and $r$, and a time from which on that set does not face any communication or server failure, then server $r$ eventually ZDelivers action $a$.*

$$\Diamond(\exists a_{s,i} \wedge stable\_system(s,r)) \Rightarrow \Diamond \exists a_{r,i}.$$

*Proof.* Since there exists a set of servers containing $s$ and $r$, and a time from which on that set does not face any communication or process failures then, according to Assumption 1 on the group communication, there is a time at which the group communication delivers a view change $v$ containing $s$ and $r$ to both $s$ and $r$, and from this time on does not deliver any other view change to $s$ and $r$.

If $a$ is ordered at $r$ at the time $v$ is delivered to $r$ then, according to Theorem 1, $r$ orders $a$ at the same order $s$ ordered $a$.

Assume $a$ is not ordered at $r$ at the time $v$ is delivered to $r$. There are only two cases:

*Server $s$ ordered action $a$ before the delivery of $v$.* In this case, after $v$ is delivered, according to the algorithm, $s$ and $r$ send State messages, and exchange actions and knowledge (see CodeSegment 5.6). Since $s$ already ordered action $a$, the most updated server already ordered $a$. Moreover, according to Theorem 1, $a$ was ordered at the most updated server at the same order as in $s$. According to Assumption 2 on the group communication, all these messages are delivered to $r$. Therefore, $a$ and its order are eventually delivered to $r$. According to the algorithm, $r$ orders $a$ (*OR-3*). According to Theorem 1, $r$ orders $a$ at the same order $s$ ordered $a$.

*Server $s$ ordered action $a$ after the delivery of $v$.* Therefore, since no other view is delivered after $v$, a primary component *PC*(*px*) is created within $v$, with s and r as members. Since there are no communication or server failures, and by Assumption 2 on the group communication layer, both $s$ and $r$ eventually install *PC*(*px*). Since $s$ ordered action $a$ at that primary component, only three sub-cases are possible:

- Server $s$ ordered $a$ according to *OR-1.2*. Therefore, $a$ was a yellow action at $s$ which was ordered when installing *PC*(*px*) (see CodeSegment 5.10). According to Assumption 2 on the group communication, eventually, $r$ has the same set of actions after the retransmissions and gets all the CPC messages. Therefore, $r$ eventually installs *PC*(*px*). According to the algorithm, when installing, $r$ orders its yellow actions, including action $a$. According to Theorem 1, $r$ orders $a$ at the same order $s$ ordered $a$.
- Server $s$ ordered $a$ according to *OR-2*. Therefore, $a$ was a red action at $s$ which was ordered when installing *PC*(*px*) (see CodeSegment 5.10). According to Assumption 2 on the group communication, $r$ eventually installs *PC*(*px*), and $r$ also has the same set of actions after the retransmissions. According to the algorithm, $r$ also orders the red action $a$ according to *OR-2*. According to Theorem 1, $r$ orders $a$ at the same order $s$ ordered $a$.
- Server $s$ ordered $a$ according to *OR-1.1*. Therefore $a$ is delivered to $s$ in view $v$ (see CodeSegment 5.2). Since there are no communication or server failures, $a$ is also deliverable at the group communication layer of $r$. According to assumption 2 on the group communication layer, $a$ is eventually delivered to $r$. According to the algorithm $r$ immediately orders $a$, and according to Theorem 1, it orders $a$ at the same order $s$ ordered $a$.

□

The above property in itself guarantees progress of the group of servers as long as at least one server orders an action. The following liveness property will describe the conditions under which an action submitted for ordering by a client is actually ZDelivered by at least one server. The progress relies on the *eventual path*[6] concept that was introduced in section 2.2.

**Theorem 4 (Liveness).** *If server $s$ receives a ZBCast($m$) invocation from a client, and there exists a primary component PC(px) which does not face any communication or server failures and a server $r$ that installed PC(px) such that $m$ is in the causal past of $r$, then $r$ orders (ZDelivers) $m$.*

*Proof.* According to the algorithm, $s$ creates an action $a$ corresponding to $m$ and SafeBCasts it to the current component after saving the action on permanent storage (CodeSegments 5.1, 5.2 and 5.8).

If $s$ does not crash, according to the Self-Delivery property of the group communication (Property 6 in the Appendix), $s$ receives $a$ from the group communication and places it in its *actionQueue* (Code Segment 5.14). If $s$ receives $a$ in a primary component, $s$ ZDelivers $a$ according to OR-1.1. In this case, $r$ is actually $s$.

Let $r$ be the first server on the eventual propagation path (in the causal future) of $a$ that installs a primary component $PC(px)$. If $s$ receives $a$ in a transitional view following a primary component, $s$ marks $a$ as yellow. According to the algorithm, since $PC(px)$ is the first primary component in $a$'s causal future, $a$ is propagated as yellow until $r$ receives it before installing $PC(px)$. At that point, $r$ ZDelivers $a$ as it installs $PC(px)$ at OR-1.2.

If $s$ receives $a$ in a non-primary component, it marks it as red. According to the algorithm, $r$ receives $a$ as red upon completing the ExchangeActions procedure and before installing

[6] The use of eventual path propagation also guarantees the optimality of our liveness properties. Intuitively, the ZBCast algorithm makes the maximum possible progress as long as **any** pair of servers is connected for sufficient amount of time. This assesment does not evaluate the optimality of the information exchange procedures.

$PC(px)$. According to the algorithm, $r$ ZDelivers $a$ when installing $PC(px)$ (OR-2).

If $s$ crashes before receiving $a$ from the group communication and subsequently recovers, $s$ executes the Recover procedure. At this time $s$ goes through the messages in its *ongoingQueue* which was saved on stable storage, finds $a$ and marks it as Red and places it in its *actionQueue* (CodeSegment 5.13). According to the algorithm, $r$ receives $a$ and its order before installing the $PC(px)$. and orders it according to OR-2 before installing $PC(px)$.

□

## 6 Dynamic Server Instantiation and Removal

In this section we present an extension to the PGTO algorithm presented in Section 5. As mentioned in the description of the model, the ZBCast algorithm works under the limitation of a fixed set of servers which needs to be known in advance. It is of great value, however, to allow for the dynamic addition of new servers as well as for their removal. This feature does not only increase the flexibility of the system setup, but can also improve the system performance and liveness. If the system does not support permanent removal of replicas, it is susceptible to blocking in case of a permanent failure or long-term disconnection of a majority of nodes in the primary component.

### *6.1 Algorithm Description*

Dynamically changing the set of servers is not straightforward: the set change needs to be synchronized over all the participating servers in order to avoid incorrect behaviour such as two distinct components deciding they are the primary, one being the rightful one in the old configuration, the other being entitled to this in the new configuration. Since this is basically a consensus problem, it cannot be solved in a traditional fashion. We circumvent the problem with the help of the persistent global total order that the algorithm provides.

When a server *r* wants to permanently leave the system, it broadcasts a PERSISTENT_LEAVE message that is ordered as if it was an action message. When this message becomes *green* at server *s*, *s* can update its local data structures to exclude *r* from the list of potential servers. The PERSISTENT_LEAVE message can also be administratively inserted into the system to signal the permanent removal, due to failure, of one of the servers. The message can be issued by a site that is still in the system and contains the server id of the dead node.

Adding a new server to the ZBCast group is, in principle, a similar operation and relies on the existing system ordering a PERSISTENT_JOIN message, which determines the point in the message ordering history from which on the new server is required to ZDeliver actions together with the rest of the group. However, the operation is more delicate due to the persistence property of the ZDelivery. From the Global Total Order perspective, in order to extend the correctness property to the new server $r$, $r$ only needs to set its green line (the global total order counter) to the value assigned by the old system to the PERSISTENT_JOIN message. From the persistency perspective however, it might be expected that the $r$ holds an action-log including actions ordered before its introduction to the group which may have been missed by other servers in the system. In practice, when $r$ joins the system it needs to request the transfer of all actions starting from the "white line" of the system up to the current state. This requirement may become even stricter if we take into account application requirements. An example and discussion are provided in section 7 where we present a database replication service that employs the ZBCast primitive.

**CodeSegment 6.1** Online reconfiguration in the ZBCast algorithm

```
MarkGreen (Action)
1   MarkRed(Action)
2   if (Action not green)
3      place Action just on top of the last green action
4      greenLines[ serverId ] = Action.action_id
5        if (Action.type == PERSISTENT_JOIN && Action.join_id
not in local structures)
6          extend greenLines, redCut to include new server id
7          greenLines[Action.join_id] = Action.action_id
9          if (Action.action_id == serverId)
10            start action-log transfer to joining site
11         elsif (Action.type == PERSISTENT_LEAVE & & Ac-
tion.leave_id is in local structures)
12         reduce greenLines, redCut to exclude Action.leave_id
13         if (Action.leave_id == serverId) exit
14     else
15         ZDeliver( Action )
When new server initiates connection
16   if (state == RegPrim) or (state == NonPrim)
17     if (new server not in local data structures)
18       create PERSISTENT_JOIN action
19       SafeBCast(action)
20     else
21       continue action-log transfer to joining site
When server wants to leave the system
22    if (state == RegPrim) or (state == NonPrim)
23      create PERSISTENT_LEAVE action
24      SafeBCast(action)
```

**CodeSegment 6.2** Joining the ZBCast group

```
25  while not updated
26    (re)connect to server in the system
27    transfer action-log
28  set greenLines[serverId] to the action_id given in the system to
the PERSISTENT_JOIN action.
29  state = NonPrim
30  join ZBCast group and start executing ZBCast algorithm.
```

Algorithm 6.1 shows the modifications that need to be added to the ZBCast algorithm described in Section 5 to support online reconfiguration. We preserve the notations from the previous section. Algorithm 6.2 shows the actions that need to be performed by the joining site before it can join the ZBCast group and start executing the algorithm.

A new server *r* that wants to join the ZBCast group will first need to connect to one of the members (*s*) of the system, without yet joining the group. *s* will act as a representative for the new site to the existing group by creating a PERSISTENT_JOIN message to announce *r*'s intention to join the group. This message is ordered as a regular action, according to the standard algorithm. When the message becomes *green* at a server, that server updates its data structures to include the newcomer's server id and set the green line (the last globally ordered message that the server has) for the joining member as the action corresponding to the PERSISTENT_JOIN message. Basically, from this point on the servers acknowledge the existence of the new member, although *r* is still not connected to the group. When the PERSISTENT_JOIN message becomes green at the peer server (the representative), the peer server will take a snapshot of the action log and start transferring it to the joining member, starting from the action marking the white line of the system up to the action corresponding to the PERSISTENT_JOIN message that introduces $r$. If the initial peer fails or a network partition occurs before the transfer is finished, the new server will try to establish a connection with a different member of the system and continue its update. If the new peer already ordered the PERSISTENT_JOIN message sent by the first representative, it knows about *r* and the state that it has to reach before joining the system, therefore is able to resume the transfer procedure. If the new peer has not yet ordered the PERSISTENT_JOIN message, it issues another PERSISTENT_JOIN message for *r*. PERSISTENT_JOIN messages for members that are already present in the local data structures are ignored by the existing servers, so only the first ordered PERSISTENT_JOIN defines the entry point of the new site into the system. Since the algorithm guarantees global total ordering, this entry point is uniquely defined. Finally, when the transfer is complete, *r* sets the action counter to the last action that was ordered by the system and joins the group. This is seen as a view change by the existing members which go through the two EXCHANGE states and continue according to the algorithm.

### *6.2 Proof of Correctness*

The algorithm in its static form was proven correct in the previous section. Here we prove that the enhanced dynamic version of the algorithm still preserves the same guarantees.

The properties specified by Theorems 1 and 2 need to be refined to encompass the removal of servers or the addition of new servers to the system.

**Theorem 5 (Global Total Order (dynamic)).** *If both servers s and r ZDelivered their ith action, then these actions are identical.*

*Proof.* Consider the system in its start-up configuration set. Any server in this configuration will trivially maintain this property according to Theorem 1. Consider a server *s* that joins the system. The safety properties of the static algorithm guarantee that after ordering the same set of actions, all servers will have the same consistent action-log. This is the case when a PERSISTENT_JOIN action is ordered. According to the algorithm *s* sets its global action counter to the one assigned by the system to the PERSISTENT_JOIN action. From this point on the behavior of *s* is indistinguishable from a server in the original configuration and the claim is maintained as per Theorem 1. □

**Theorem 6 (Global FIFO Order (dynamic)).** *If server r ZDelivered an action a generated by server s, then r already ZDelivered every action that s generated prior to a, after r joined the system.*

*Proof.* According to Theorem 2, the theorem holds true from the initial starting point until a new member is added to the system. Consider *r*, a member who joins the system. From Theorem 5, the PERSISTENT_JOIN message is ordered at the same place at all servers. All actions generated by *s* and ordered after the PERSISTENT_JOIN message are ordered similarly at every server, including *r*, according to Theorem 5. Since Theorem 2 holds for any other member, this is sufficient to guarantee that *r* orders all other actions generated by *s* prior to a, and ordered after *r* joined the system. □

Similarly, the liveness properties specified in section 5.2 need to be refined to include the notion of dynamic configuration. Theorem 4 is trivially satisfied by the dynamic algorithm without any change, as it relies on the causal history of an action throughout its life in the system, notion which doesn't change in the dynamic context.

**Theorem 7 (ZDelivery propagation (dynamic)).** *If server s orders action a in a configuration that contains r and there exists a set of servers containing s and r, and a time from which on that set does not face any communication or process failures and r does not leave and is not removed from the server set, then server r eventually orders action a.*

*Proof.* The theorem is a direct extension of Theorem 3, which acknowledges the potential existence of different server-set configurations. On a static configuration, an action that is ordered by a server is ordered by all servers in the same configuration as a direct consequence of the static liveness property. In the presence of configuration changes, the eventual path propagation is unaltered since any new member will join they system with a complete log of all the actions that might be needed by another server. Servers that leave the system or crash do not meet the requirements for the liveness property, while servers that join the system will order the actions generated in any configuration that includes them, unless they crash. □

## 7 From ZBCast to Database Replication

We looked at the Persistent Global Total Ordering problem, not just from a theoretical perspective, but motivated by a very important application that requires both a strong, provably correct solution and high performance: database replication.

The state machine approach [28] to database replication ensures that replicated databases that start consistent will remain consistent as long as they apply the same deterministic actions (transactions) in the same order. Thus, we can reduce the database replication problem to the problem of constructing a persistent global total order of actions.

In this section we describe a database replication model using the ZBCast primitive and discuss its benefits and limitations.

### *7.1 Replication Model*

A *Database* is a collection of organized, related data. Clients access the data by submitting *transactions.* consisting of a set of commands that follow the ACID properties. A replication service maintains a replicated database in a distributed environment. Each server from the server set maintains a private copy of the database. The initial state of the database is identical at all servers. Several models of consistency can be defined, the strictest of which is *one-copy serializability* that requires that the concurrent execution of transactions on a replicated data set is equivalent to a serial execution on a non-replicated data set. We focus on enforcing the strict consistency model, but we can also support weaker models.

An *action* defines a transition from the current state of the database to the next state; the next state is completely determined by the current state and the action. We view actions as having a *query* part and an *update* part, either of which can be missing. Each replica is part of the same ZBCast group and client transactions are encapsulated into messages which are ordered using the ZBCast primitive. The basic model best fits non-interactive transactions, but in Section 7.3 we show how other transaction types can also be supported.

This model assumes that all the replicas execute transactions deterministically and the outcome of the same transaction is identical at all replicas if submitted to the same database state. Since all transactions are executed serially on each replica, there is no risk of transaction abort due to deadlocks or conncurrency control. Our system can deal with transient commit failures at one of the copies, by detecting the failure through the database response and retransmitting the request until it succeeds. However, if a replica consistently cannot commit a transaction that was ordered by the ZBCast algorithm, it will stop making progress and stop responding to the ZBCast algorithm requests resulting in it being detected as crashed by the group communication layer. Ultimately this replica needs to be eliminated from the configuration, unless the problem can be externally fixed, as our model does not allow for non-deterministic transaction abort.

Under these assumptions, we can transparently employ the ZBCast algorithm in order to provide database replication. A database client connects to an interceptor module that transforms the transaction request into a ZBCast request for the PGTO layer. The message is ordered and upon ZDelivery, the interceptor applies it to the database and returns the results to the requesting client.

### *7.2 Dynamic Replication*

As we anticipated in section 6.1, supporting a dynamic set of replicas requires additional synchronization before a new replica can join the system. In fact, the new replica needs to be dynamically synchronized with the online replicated databases, up to the state defined by the action that signaled the joining intent (the PERSISTENT_JOIN action). Our suggested online reconfiguration method allows the replicated system to continue execution while the joining replica is updated. According to the ZBCast reconfiguration algorithm 6.1, the new replica is updated while the existing system continues execution. When the new replica joins the replicated group, a view change is detected and the final update is performed as guided by the regular ZBCast algorithm. In order to further optimize the process, several incremental synchronization points may be determined by the peer server responsible for updating the new member, which will reduce the synchronization amount needed during the Exchange states of the ZBCast algorithm.

The online reconfiguration problem for database replication is of well known complexity. [23] provides an indepth study of the various techniques that can be used for this purpose and their respective drawbacks and presents a complete solution based on a different group membership paradigm. While the described method requires the joining site to be permanently connected to the primary component while being updated it provides the explicit framework that gives control to the application (in this case the database) over deciding when the synchronization is complete so that the new replica can join the system. We maintain the flexibility of the ZBCast algorithm by allowing joining servers to be connected to non-primary components during their update stage. It can even be the case that a new site is accepted into the system without **ever** being connected to the primary component, due to the eventual path propagation method. The insertion of a new server into the system in a non-primary component, could be useful to certain applications.

### *7.3 Design Tradeoffs*

From a design perspective, we can distinguish between three possible approaches to the architecture of a replicated database. Our model follows the *black box* approach which does not assume any knowledge about the internal structure of the database system or about the application semantics. This allows for completely transparent replication as neither the client nor the database need to be modified in order to support the replication. Also, this architecture enables the replication system to support heterogeneous replication where different database managers from different vendors replicate the same logical database.

In contrast, a *white box* approach such as [21] will integrate the replication mechanism within the database itself, attempting to exploit the powerful mechanisms (concurrency control, conflict resolution) that are implemented inside the database, at the price of losing transparency. A middle-way *gray box* approach [19] assumes that the database system is enhanced by providing additional primitives that can be used from the outside but does not include the replication mechanism inside the database itself. [19] also exploits the data partitioning common in many databases, assuming that the application can provide information about the conflict classes that are addressed by each transaction. This approach also permits the use of row replication, where a transaction is executed on just one database and its effect is replicated to the other servers. This reduces the load on each database server

but may increase significantly the network load as the modified row data can be significantly larger than the standard SQL transaction.

We argue that although our design does not allow, in its basic form, concurrent transaction execution, it doesn't suffer a performance drawback because it uses an efficient synchronization algorithm. Furthermore, even a highly concurrent synchronous replication system cannot overcome the fundamental cost of global synchronization needed by any PGTO algorithm and would be therefore limited by the performance of the synchronization module. Section 8 presents some performance measurements that support this argument.

In the presentation of our model we mention that our replication architecture assumes that each (possibly multi-operation) transaction is deterministic and can be encapsulated in one action, thus removing the possibility of executing normal interactive transactions. This assumption can be relaxed to a certain degree. We can allow, for example, a transaction to execute a deterministic procedure that is specified in the body of the transaction and that depends solely on the current state of the database. These transactions are called active transactions.

With the help of active transactions, one can mimic interactive transactions where a user starts a transaction, reads some data then makes a user-level interactive decision regarding updates. Such transactions can be simulated with the help of two actions in our model. The first action contains the query part of the transaction. The second action is an active action that encapsulates the updates dictated by the user, but first checks whether the values of the data read by the first action are still valid. If they are not valid, the second action is aborted, as if the transaction was aborted in the traditional sense. Note that if one server aborts, all of the servers abort that (trans)action since they apply an identical deterministic rule to an identical state of the database, as guaranteed by the algorithm.

An extended discussion on how various semantics and transaction types can be supported by a solution based on the ZBCast algorithm can be found in [6], while [2] presents and evaluates a practical prototype that provides replication for a PostgreSQL database system.

## 8 Performance Analysis

In this section we evaluate the performance of the PGTO algorithm and compare it with two solutions that provide similar guarantees: two-phase commit (2PC) and COReL by Keidar [20]. While the 2PC comparison seems unrelated with the PGTO algorithm, it is the solution adopted by most replicated database systems that require strict consistency and the comparison becomes relevant from that perspective. Since it is a well defined algorithm, with well understood performance trade-offs, 2PC also helps highlight the impact of some of the design choices made by our solution. In effect, 2PC requires two forced disk writes and 2n unicast messages per action. COReL exploits group communication properties to provide PGTO and requires one forced disk write and n multicast messages per action. In contrast, our algorithm only requires 1/n forced disk writes and one multicast message per action on average (only the initiating server needs to force the action to disk).

We implemented COReL using the same codebase with our algorithm, the only modifications being due to the protocol differences. We also implemented 2PC following the standard algorithm specifications. Our 2PC implementation allows for actions submitted by different clients to be commited concurrently. This leads to a better 2PC performance than if it would implement PGTO and is also optimistic with respect to database requirements, where possible locking would prevent the full concurrency allowed by our implementation. For this reason we refer to our 2PC implementation as "upper-bound 2PC".

Since we are interested in the intrinsic performance of the algorithms. clients receive responses to their actions as soon as the actions are globally ordered, without any interaction with a database. The tests conducted measure the performance that updates would experience in a replicated system, during normal operation when no network events occur. A follow-up work [2] evaluates a complete solution that replicates a Postgres database over local and wide area networks using the ZBCast algorithm and the method described in this paper.

All the tests were conducted with 14 replicas, each running on a dual processor Pentium III-667 with Linux connected by a 100Mbits/second local area switch. Each message is 200 bytes long.

Figure 5(a) compares the maximal throughput that a system of 14 servers can sustain under each of the three methods. We vary the number of clients that simultaneously submit requests into the system between 1 and 28, evenly spread between the servers as much as possible. The clients are constantly injecting actions into the system, the next action from a client being introduced immediately after the previous action from that client is completed and its result reported to the client.

Our engine achieves a maximum throughput of 1050 actions/second once there are sufficient clients to saturate the system, outperforming the other methods by at least a factor of 10. COReL outperforms the uppper-bound 2PC as expected, mainly due to the saving in disk writes reaching a maximum of 110 actions per second as opposed to 63 actions per second for the upper-bound 2PC. The results reflect the impact of additional forced disk write operations required by each solution. 2PC pays the highest price and the nodes are limited by the number of forced disk writes that they can perform per second, which in a LAN setting has much higher weight than the latency and bandwidth considerations.

In order to estimate the impact on performance of forced writes to disk, we used asynchronous disk writes instead of forced disk writes for a set of tests[7]. Figure 5(b) shows that our engine tops at processing 3000 actions/second. Under the same conditions, the upper-bound 2PC algorithm achieves 400 actions/second. COReL reaches a throughput of approximately 100 actions/second with 28 clients, but more clients are needed in order to saturate the COReL system due to its higher latency. With 50 clients, CORel saturates the system

[7] This test is relevant for evaluating the potential performance such algorithms would have in a database replication architecture which uses high-performance storage technology such as flash disks.

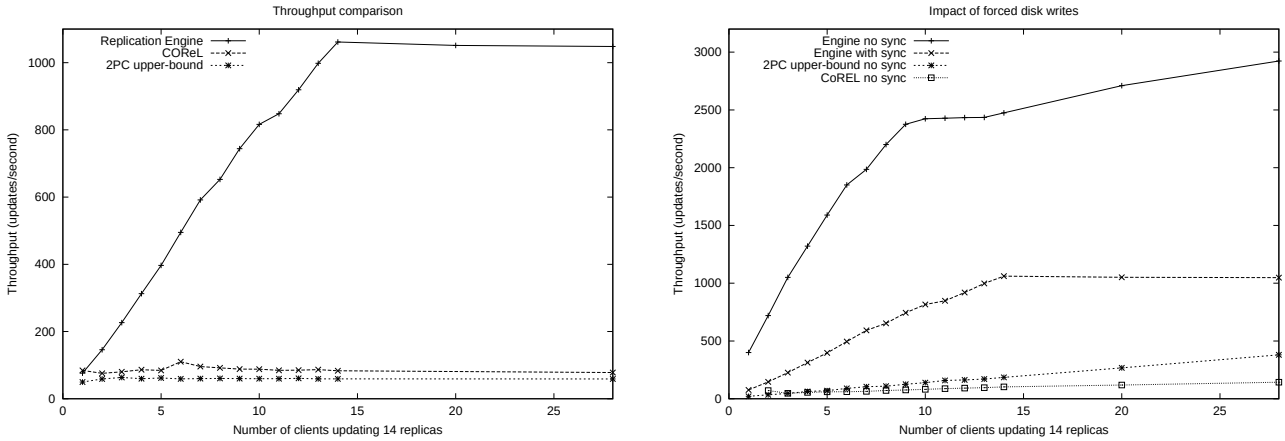


**Fig. 5.** Throughput Comparison

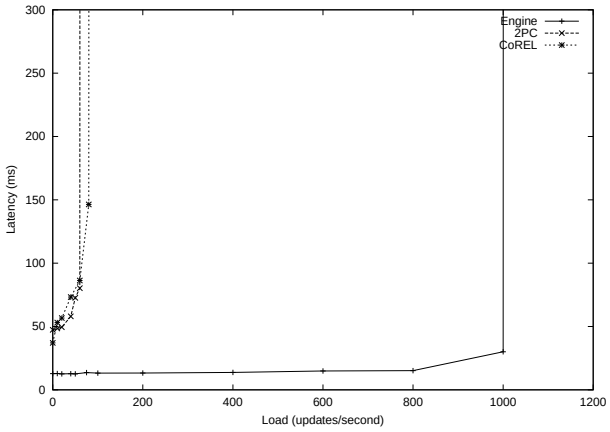


**Fig. 6.** Latency variation with load

with about 200 actions/second. 2PC outperforms COReL in this experiment because of two reasons: the fact that we use an upper-bound 2PC as mentioned above, and the particular switch that serves our local area network that is capable of transmitting multiple unicast messages between different pairs in parallel favoring the 2PC style communication.

We also measured the response time a client experiences under different loads (Figure 6). Our engine maintains an average latency of 15ms with load increasing up to 800 actions/second and breaks at the maximum supported load of 1050 actions/second. COReL and 2PC experience latencies of 35ms up to 80ms under loads up to 100 actions/second with COReL being able to sustain more throughput.

## 9 Related Work

The work presented in this paper extends standard group communication techniques to build a new group broadcast primitive and shows how this primitive can be applied to provide peer synchronous database replication. In this section we acknowledge research that spans these areas and relates to one or several aspects discussed in the present work.

Group Communication has a long history and a number of systems exist that implement different specifications. ISIS [11], Phoenix [24] are among those that employ a primary membership service, while Transis [3], Horus [32], Totem [7], RMP [33] and Spread [5] employ a partitionable membership service. The systems also provide different delivery properties among which Virtual Synchrony [10] and Extended Virtual Synchrony [25] are of direct importance to this work as seen in sections 3 and 5.

These specifications provide delivery guarantees such as Total Ordering which are maintained only with respect to single views. Fekete et al [15] specify a partitionable group communication service called View Synchrony and provide an algorithm that builds a Global Total Order on top of this service. The conceptual solution is related to ours as it borrows from [20] and [4], but does not cope with the ”partial amnesia” problem assuming that processors preserve their state between crashes and recoveries, and also considers a only static set of participants.

In order to provide global total order we define a primary-view membership on top of a partitionable-view membership group communication. [27] specifies a primary-membership group communication based on a dynamic quorum and present a total order algorithm on top of the new specification. The authors decouple the total order from the primary-membership specification thus separating our Exchange states by abstracting them as application tasks and defining an interface through which the application informs the group communication that the information exchange is completed (the application ”registers” the view).

Atomic Broadcast [17] in the context of Virtual Synchrony [10] emerged as a promising building block for database replication solutions. Several algorithms were introduced [31, 26] to implement replication solutions based on total ordering. All these approaches, however, work only in the context of non-partitionable environments.

Keidar [20] uses the Extended Virtual Synchrony (EVS) [25] model to propose an algorithm that supports network partitions and merges. The algorithm requires that each transaction message is end-to-end acknowledged, even when failures are not present, thus increasing the latency of the protocol.

Kemme, Bartoli and Babaoglu[23] study the problem of online reconfiguration of a replicated system in the presence of network events, which is an important building block for

a replication algorithm. They propose various useful solutions to performing the database transfer to a joining site and provide a high-level description of an online reconfiguration method based on Enriched Virtual Synchrony allowing new replicas to join the system if they are connected with the primary component. Our solution can be coupled with these database transfer techniques and adds the ability to allow new sites to join the running system without the need to be connected to the primary component.

Kemme and Alonso [22] present and prove the correctness for a family of replication protocols that support different application semantics. The protocols are introduced in a failure-free environment and then enhanced to support server crashes and recoveries. The model does not allow network partitions, always assuming that disconnected sites have crashed. In their model, the replication protocols rely on external view-change protocols that provide uniform reliable delivery in order to provide consistency across all sites. Our work shows that the transition from the group communication uniform delivery notification to the strict database consistency is not trivial, we provide a detailed algorithm for this purpose and prove its correctness.

Finally, this work builds on [6] where we exposed the delicacy of implementing a persistent global total order service in a network model that includes network partitions and remerges and allows servers to crash and recover under the same identifier. The current work structures the ideas presented in [6] by introducing a new ordering primitive and separating it from the database replication application and provides the full correctness proof for the algorithm.

## 10 Conclusions

We introduced a group communication primitive called ZBCast that provides Persistent Global Total Ordering in a generic environment that supports server crashes and recoveries exhibiting *partial amnesia* - identity preservation but loss of any state information that is not saved on permanent storage. Also the system supports network partitions and recoveries and continues to make progress even under these conditions. We presented a complete ZBCast algorithm highlighting why Total Order is not sufficient to provide the desired guarantees and proving the correctness our solution.

In order to emphasize the utility of a ZBCast primitive, we showed how it can be employed to provide fully-synchronous, active database replication. We presented an extension of the algorithm that supports dynamic instantiations and removals of replicas and sketched its correctness proof. We finally evaluated our algorithm against other solutions providing PGTO and we show the benefits of our optimized solution that fully exploits the group communication power.

*Acknowledgements.* We thank the journal editor and the anonymous reviewers for their extensive comments and insight which helped us focus the paper. We also thank Michel Raynal for his advice towards building a bridge between various system models in distributed algorithms.

## References

1. O. Amir, Y. Amir, and D. Dolev. A highly available application in the Transis environment. *Lecture Notes in Computer Science*, 774:125–139, 1993.
2. Y. Amir, C. Danilov, M. Miskin-Amir, J. Stanton, and C. Tutu. On the performance of consistent wide-area database replication. Technical Report CNDS 2003-3, Johns Hopkins University, Center for Networking and Distributed Systems, 2003.
3. Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A communication sub-system for high availability. In $22^{nd}$ *Annual International Symposium on Fault-Tolerant Computing (FTCS)*, pages 76–84, Boston, Massachusetts, USA, July 1992. IEEE Computer Society.
4. Y. Amir, D. Dolev, P. M. Melliar-Smith, and L. E. Moser. Robust and efficient replication using group communication. Technical Report CS94-20, The Hebrew University of Jerusalem, Institute of Computer Science, November 1994.
5. Y. Amir and J. Stanton. The spread wide area group communication system. Technical Report CNDS 98-4, Johns Hopkins University, Center for Networking and Distributed Systems, 1998.
6. Y. Amir and C. Tutu. From total order to database replication. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*, pages 494–503, Vienna, Austria, July 2002. IEEE.
7. Yair Amir, L. E. Moser, P. M. Melliar-Smith, D.A. Agarwal, and P. Ciarfella. The totem single-ring ordering and membership protocol. *ACM Transactions on Computer Systems*, 13(4):311–342, November 1995.
8. R. Baldoni and M. Raynal. Fundamentals of distributed computing: A practical tour of vector clock systems. *IEEE Distributed Systems Online*, 3(2), 2002.
9. T. Basten. Breakpoints and time in distributed computations. In *Workshop on Distributed Algorithms*, pages 340–354, 1994.
10. K. P. Birman and T. A. Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the ACM Symposium on OS Principles*, pages 123–138, Austin, TX, 1987.
11. Keneth P. Birman and Robert V. Renesse. *Reliable Distributed Computing with the Isis Toolkit*. IEEE Computer Society Press, March 1994.
12. Gregory V. Chockler, Idit Keidar, and Roman Vitenberg. Group communication specifications: A comprehensive study. *ACM Computing Surveys*, 33(4):427–469, December 2001.
13. X. Defago, A. Schiper, and P. Urban. Totally ordered broadcast and multicast algorithms: a comprehensive survey. Technical Report DSC/2000/036, Dept. of Communication Systems, EPFL, 2000.
14. The Ensemble Distributed Communication System. http://www.cs.cornell.edu/Info/Projects/Ensemble/.
15. A. Fekete, N. Lynch, and A. Shvartsman. Specifying and using a partitionable group communication service. *ACM Transactions on Computer Systems*, 19(2):171–216, May 2001.
16. M. H. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.
17. V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In Sape Mullender, editor, *Distributed Systems*, chapter 5. Addison-Wesley, second edition, 1993.
18. S. Jajodia and D. Mutchler. Dynamic voting algorithms for maintaining the consistency of a replicated database. *ACM Transactions on Database Systems*, 15(2):230–280, 1990.
19. R. Jimenez-Peris, M. Patino-Martinez, B. Kemme, and G. Alonso. Improving the scalability of fault-tolerant database

clusters. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*, pages 477–484. IEEE, July 2002.
20. I. Keidar. A highly available paradigm for consistent object replication. Master's thesis, Institute of Computer Science, The Hebrew University of Jerusalem, Israel, 1994.
21. B. Kemme and G. Alonso. Don't be lazy, be consistent: Postgres-R, a new way to implement database replication. In *Proceedings of the* $26^{th}$ *International Conference on Very Large Databases (VLDB)*, Cairo, Egypt, September 2000.
22. B. Kemme and G. Alonso. A new approach to developing and implementing eager database replication protocols. *ACM Transactions on Database Systems*, 25(3):333 – 379, 2000.
23. B. Kemme, A. Bartoli, and Ö. Babaoğlu. Online reconfiguration in replicated databases based on group communication. In *Proceedings of the Internationnal Conference on Dependable Systems and Networks*, Göteborg, Sweden, 2001.
24. C. P. Malloth, P. Felber, A. Schiper, and U. Wilhelm. Phoenix: A toolkit for building fault-tolerant distributed applications in large scale. In *Workshop on Parallel and Distributed Platforms in Industrial Products*, San Antonio, Texas, USA, October 1995. Workshop held during the 7th IEEE Symp. on Parallel and Distributed Processing, (SPDP-7).
25. L. E. Moser, Y. Amir, P. M. Melliar-Smith, and D. A. Agarwal. Extended virtual synchrony. In *International Conference on Distributed Computing Systems*, pages 56–65, 1994.
26. F. Pedone, R. Guerraoui, and A. Schiper. Exploiting atomic broadcast in replicated databases. In *Proceedings of EuroPar (EuroPar'98)*, September 1998.
27. R. De Prisco, A. Fekete, N. Lynch, and A. Shvartsman. A dynamic view-oriented group communication service. In *Symposium on Principles of Distributed Computing*, pages 227–236, 1998.
28. F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.
29. John Schultz. Partitionable virtual synchrony using extended virtual synchrony. Master's thesis, Department of Computer Science, Johns Hopkins University, January 2001.
30. The Spread Toolkit. http://www.spread.org.
31. I. Stanoi, D. Agrawal, and A. El Abbadi. Using broadcast primitives in replicated databases. In *Proceedings of the* $18^{th}$ *IEEE International Conference on Distributed Computing Systems '98*, pages 148–155, Amsterdam, May 1998.
32. R. van Renesse, K. P. Birman, and Silvano Maffeis. Horus: A flexible group communication system. *Communications of the ACM*, 39(4):76–83, April 1996.
33. B. Whetten, T. Montgomery, and S. Kaplan. A high performance totally ordered multicast protocol. In *Theory and Practice in Distributed Systems, International Workshop*, Lecture Notes in Computer Science, page 938, September 1994.

## A Extended Virtual Synchrony

The Group Communication System is assumed to support Extended Virtual Synchrony semantics as defined below. This set of properties is largely based on the popular survey in [12] and the definition of related semantics in [25] and [29].

We define that some event occurred in view $v$ at process $p$ if the most recent view installed by process $p$ before the event was $v$.

1. *Self Inclusion*
   If process $p$ installs a view $v$ then $p$ is a member of $v$.
2. *Local Monotonicity*
   If process $p$ installs a view $v$ after installing a view $v'$ then its identifier $id_v$ is greater than $v'$'s identifier $id_{v'}$.
3. *Same View Delivery*
   If processes $p$ and $q$ both deliver a message $m$, then they both deliver $m$ in the same view.
4. *Delivery Integrity*
   1) If process $p$ delivers a message $m$ in a view $v$, then there exists a process $q$ that sent $m$ in $v$ causally before $p$ delivered $m$.
   2) If process $p$ delivers a message $m$ in a regular view $v$ or in the transitional view $v'$ immediately following $v$, there exists a process $q$ that sent $m$ in $v$.
5. *No Duplication*
   A message is sent only once. A message is delivered only once to the same process.
6. *Self Delivery*
   If process $p$ sends a message $m$, then $p$ delivers $m$ unless it crashes.
7. *Transitional Set*
   1) Every process is part of its transitional set for a view $v$.
   2) If a process $p$ installs a view in a previous view, then the transitional set for the new view at $p$ is a subset of the intersection between the two views' membership sets.
   3) If two processes $p$ and $q$ install the same view $v$ then $q$ is included in $p$'s transitional set for $v$ if and only if then $p$ and $q$ have the same previous view.
   4) If processes $p$ and $q$ install the same view in the same previous view, they have the same transitional sets in their new views.
8. *Virtual Synchrony*
   Two processes $p$ and $q$ that move together[8] through two consecutive views $v$ and $v'$ deliver the same set of messages in $v$.
9. *FIFO Delivery*
   If message $m$ is sent before message $m'$ by the same process in the same view, then any process that delivers $m'$ delivers $m$ before $m'$.
10. *Causal Delivery*
    If message $m$ causally precedes message $m'$ and both are sent in the same view, then any process that delivers $m'$ delivers $m$ before $m'$.
11. *Agreed Delivery*
    1) Agreed delivery maintains all causal delivery guarantees.
    2) If agreed messages $m$, and later, $m'$ are delivered by process $p$, and $m$ and $m'$ are also delivered by process $q$, then $q$ delivered $m$ before $m'$.
    3) If agreed messages $m$, and later, $m'$ are delivered by process $p$ in view $v$, and $m'$ is delivered by process $q$ in $v$ before a transitional signal, then $q$ delivers $m$. If messages $m$, and later, $m'$ are delivered by process $p$ in view $v$, and $m'$ is delivered by process $q$ in $v$ after a transitional signal, then $q$ delivers $m$ if $r$, the sender of $m$, belongs to $q$'s transitional set.
12. *Safe Delivery*
    1) Safe delivery maintains all agreed delivery guarantees.
    2) If process $p$ delivers a safe message $m$ in view $v$ be-

[8] If processes $p$ and $q$ both install the same view in the same previous view, then $p$ and $q$ are said to move together.

fore the transitional signal, then every process $q$ of view $v$ delivers $m$ unless it crashes. If process $p$ delivers a safe message $m$ in view $v$ after the transitional signal, then every process $q$ that belongs to $p$'s transitional set delivers $m$ after the transitional signal unless it crashes.

*13. Transitional Signal*

1) Each process delivers exactly one transitional signal per view.

2) If two processes $p$ and $q$ install the same view $v$ and $q$ is included in $p$'s transitional set for $v$ then $p$ and $q$ deliver the same set of agreed messages before and after the transitional signal.