CS5320 Theory Assignment - 1

Vishal Vijay Devadiga (CS21BTECH11061)

Question

Develop a pseudo code for the SK algorithm.

Answer

Spezialetti-Kearns (SK) algorithm is an efficient and scalable algorithm for global snapshots in a distributed system. There are two phases in the SK algorithm for obtaining a global snapshot:

1. Locally recording the snapshot at every process known as efficient snapshot recording.
2. Distributing the resultant global snapshot to all the initiators known as efficient dissemination of the recorded snapshot.

SK algorithm supports concurrent initiators, efficient assembly, and distribution of a snapshot. It is based on the assumption that bidirectional channels are present in the network.

SK algorithm is as follows:

Suppose there are n processes in the distributed system and e channels.

I is the set of initiators. These initiators are the processes that initiate the global snapshot recording.

These processes first record their local snapshot and then send a marker message to all neighbors of that process with the id of the initiator.

When a process receives a marker message, it records its local snapshot and then sends a marker message to all its neighbors with the id of the initiator except the one from which it received the marker message.

Suppose a process p receives a marker message from process q. Then the master of p is the master of q, thus the initiator in that spanning tree. If p receives another marker message from suppose r with a different initiator, then the master of r is stored in the the variable `id-border-set` and is eventually sent to the master of p during spanshot collection.

Snapshot recording at a process is complete after it has received a marker along each of its channels. After this, the process sends the snapshot of that process and its children to its parent, eventually reaching the initiator.

The initiators then assembles the snapshot of that spanning tree. Then the initiators exchange information about snapshots in multiple rounds.

The process is complete when no new information is exchanged in a round.

Pseudo Code

```
merge_snapshots(snapshot1, snapshot2):
    return snapshot1 + snapshot2

merge_id_border_set(id_border_set1, id_border_set2):
    return id_border_set1 + id_border_set2

class Process
{
    id <- id of the process
    snapshot <- {}
    id_border_set <- {}
    parent <- -1
    is_initiator <- True if process is an initiator else False
    initiator <- -1 if process is not an initiator else id
    neighbors <- set of neighbors of the process

    // Record local snapshot at the process along with channel state
    record_local_snapshot():
        return local snapshot of process

    // Send marker message to process p with id of initiator
    send_marker(p):
        msg <- marker message with id of initiator
        send msg to p

    // Receive snapshot and id_border_set from process p
    // and merge it with local snapshot and id_border_set
    receive_snapshot_border(p):
        msg <- receive message from p

        // Merge received snapshot and id_border_set
        recv_snapshot <- msg.snapshot
        recv_id_border_set <- msg.id_border_set
        snapshot <- merge_snapshots(snapshot, recv_snapshot)
        id_border_set <- merge_id_border_set(id_border_set, recv_id_border_set)

    // Send a specific snapshot and id_border_set to process p
    send_snapshot_border(p, send_snapshot, send_id_border_set):
        msg <- snapshot message with send_snapshot and send_id_border_set
        send msg to p

    // Record snapshot and id_border_set at the process and its children
    record_snapshot_border():
        // Send marker to all neighbors except parent
        for each neighbor q in neighbors:
```

```
        if q != parent:
            send_marker(q)

    // Record local snapshot
    snapshot <- record_local_snapshot()

    // Receive snapshot and id_border_set from all neighbors except parent
    for each neighbor q in neighbors:
        if q == parent:
            continue
        receive_snapshot_border(q)

    // Send snapshot and id_border_set to parent
    send_snapshot_border(parent, snapshot, id_border_set)

// Receive marker messages: Runs in parallel
receive_marker():
    msg <- receive marker
    recv_init <- msg.initiator
    recv_id <- msg.id

    // If parent is not set and process is not an initiator
    if parent == -1 and is_initiator == False:
        parent <- recv_id
        initiator <- recv_init
        record_snapshot_border()
    // If parent is set and received marker from a different initiator
    else if initiator != recv_init:
        // Send the current initiator to the process that sent the marker
        send_id_border_set <- initiator
        send_snaphot <- {}
        // Send empty snapshot and id_border_set to the process that sent the marker
        send_snapshot_border(recv_id, send_snapshot, send_id_border_set)
    // If parent is set and received marker from the same initiator
    else if initiator == recv_init:
        send_id_border_set <- {}
        send_snapshot <- {}
        // Send empty snapshot and id_border_set to the process that sent the marker
        send_snapshot_border(parent, send_snapshot, send_id_border_set)

// Main function of the process
Main():
    // Execute Parallelly receive_marker to receive marker messages
    Execute Parallel
        receive_marker()
    // If process is an initiator, collect snapshot
    if is_initiator == True:
            record_snapshot_border()

    // Disseminate snapshot to all initiators
    Disseminate_Snapshot():
        // If process is an initiator
        if is_initiator == True:
            while True:
                // Flag to check if new information is exchanged
                flag <- False
                // Send/Receive snapshot, id_border_set to all neighbors
                for i in id_border_set:
                    send_snapshot_border(i, snapshot, id_border_set)
                    old_snapshot <- snapshot
                    snapshot <- merge_snapshots(snapshot, receive_snapshot_border(i))
                    // If the new snapshot is different from the old snapshot,
                    // set flag to True
                    if snapshot != old_snapshot:
                        flag <- True
                // Break if no new information is exchanged
                if flag == False:
                    break
}

// Main function
Main():
    P <- set of processes
    I <- set of initiators

    // Collect snapshots
    for each process p in P:
        p.Main()

    // Disseminate snapshots
    for each process p in I:
        p.Disseminate_Snapshot()
```

Time Complexity

It takes $O(e)$ messages to record, $O(rn^2)$ messages to assemble and distribute snapshots.

Where n is the number of processes, e is the number of channels, and r is the number of concurrent initiators.

Thus, the time complexity of the SK **algorithm** is $O(e + rn^2)$.