Theory Assignment 1

Abhay Shankar K: cs21btech11001

The following pseudocode details the Spezialetti-Kearns algorithm for distributed snapshot collection. The algorithm makes extensive use of flooding, and executes in two phases.

- The first phase is the initiation phase, where the initiator node sends out a message to all its neighbours, and waits for a response from all of them. This divides the distributed system into an implicit spanning forest, with each initiator node as the root of one of the trees. Each node bulds a snapshot of it's tree, called its *region*, by merging the responses it receives from its neighbours, which are in turn the collected snapshots of the subtrees rooted at those neighbours.
- The second phase is the dissemination phase. The responses each initiator node receives contain information about the roots of all neighbouring regions, so the initiator node broadcasts it's snapshot as well as all it's adjacent regions. This occurs over several rounds, until the snapshot stabilizes in each initiator node.

The pseudocode is similar to Rust, but is not valid Rust code. It is meant to be a demonstration of the algorithm, and is not meant to be run as is. The actual implementation would require a lot of additional code to handle the network communication, and the actual snapshot collection.

In some places, set union and set difference operations are used. These are meant to be understood as the standard set operations, and are not meant to be implemented as such in Rust. The `+` operator has been overloaded for set union, and the `-` operator for set difference.

```
fn init_snapshot(node: &Node) {
    // We are the master
    send(node.neighbours, INIT_SNAPSHOT(node.id));
    let count = 0; // #responses

    let border_set = {};
    let collected_snapshot = node.local_snapshot();

    // Wait for all responses. Initiation phase.
    while (count < node.neighbours.len()) {
        let (msg, from) = recv();
        match msg {
            INIT_SNAPSHOT(other_master) => {
                if other_master != node.id {
                    border_set.insert(other_master);
                }
                send(from, NAK);
            }
            SNAPSHOT(ss, border) => {
                count += 1;
                merge_snapshot(collected_snapshot, ss);
                border_set += border;
            }
            NAK => {
                count += 1;
            }
            _ => {}
        }
```

```
    }

    // We don't need to send to parent coz we are root.

    // Now send your stuff back out. Dissemination phase.
    let known = {node.id};

    // Loop until we have reached all regions
    while(!border_set.is_empty()) {
        send(node.neighbours, DISSEMINATE(collected_snapshot, border_set, node.id));
        //
        let wavefront = {}; // All regions at distance = round number

        // One round. We wait for every adjacent region we know of.
        while (!border_set.is_empty()) {
            let (msg, from) = recv();
            match msg {
                DISSEMINATE(ss, border, master) => {
                    border_set -= master;
                    known += master; // Everything we have already received. Note that this is not from -
                    wavefront += border;
                    merge_snapshot(collected_snapshot, ss);
                }
                _ => {}
            }
        }
        border_set = wavefront - known; // Ensures that no node is added to border_set twice.
    }
}

// We assume this function is constantly running on all nodes
fn listen(node: &Node) {
    let border_set = {};
    let dissem_rcv = {}; // To ensure termination of flooding in the dissemination stage
    loop {
        let (msg, from) = recv();
        match msg {
            INIT_SNAPSHOT(master) => {
                if node.master.is_some() {
                    send(from, NAK);
                    border_set += master;
                    continue;
                }

                // Don't have a master
                node.parent = from;
                node.master = master;

                // Record snapshot after recv mark i.e. INIT_SNAPSHOT, but before sending more messages.
                let local_snapshot = node.local_snapshot();

                // propagate
                send(node.neighbours - parent, INIT_SNAPSHOT(master));
                let count = 0;
                // Wait for all responses
                // If all NAK, leaf.
```

```
                while (count < node.neighbours.len() - 1) {
                    let (msg, from) = recv();
                    match msg {
                        NAK => {
                            count += 1;
                        }
                        SNAPSHOT(ss, border) => {
                            count += 1;
                            merge_snapshot(local_snapshot, ss);
                            border_set += border;
                        }
                    }
                }
                send(from, SNAPSHOT(local_snapshot, border_set));
            }
            DISSEMINATE(ss, border, master) => {
                // We don't have to do anything, we are not master. Just flood.
                dissem_rcv += from;
                send(node.neighbours - dissem_rcv, DISSEMINATE(ss, border, master));
            }
            _ => {}
        }
    }
}

/// This is run on the whole system.
/// Obviously, we cannot run a function on the whole system, so this is purely a demonstration.
fn take_snapshot(system: &System, thresh: f64) {
    for node in system.nodes {
        if gen_random_real() > thresh {
            init_snapshot(node); // Maybe initiator...
        } else {
            listen(node); // ...but mostly no.
        }
    }
}
```

The routines `merge_snapshot` and `Node::local_snapshot()` are assumed to be given. However, an overview:

`merge_snapshot(old, new)`

- If there are `send` events in `old` without corresponding `recv` events in `new`, add them to the appropriate channel state. Repeat for `send` events in `new` without corresponding `recv` events in `old`.
- Add all local and channel states in `new` to `old`.

We do not need to check for `recv` events without corresponding `send` events, as the algorithm is designed to ensure that this does not happen.

`Node::local_snapshot()`

From the textbook, we have the following conditions on when the local snapshot must be taken:

- Any message that is sent by a process before recording its snapshot, must be recorded in the global snapshot (from C1).
- Any message that is sent by a process after recording its snapshot, must not be recorded in the global snapshot (from C2).
- A process p_j must record its snapshot before processing a message m_{ij} that was sent by process p_i after recording its snapshot.

Which are satisfied the the pseudocode. The exact nature of the local snapshot is not important for the algorithm, as long as it satisfies these conditions.