Network Management Project

Jiannan Guo - ICT-Innovation
Niklas Semmler - ICT-Innovation

29. November 2013

1 Task

1.1 Pseudocode

Listing 1: inspired by [1]

```
messages:
(ResponseTimeArriveMessage, responseTime)
(UpdateVector, neighbor)
(TimeOut, responseTime)

objects and help functions:
A       # storing values for and performing aggregation
Comm    # communication messages
A.aggregate_local       # aggregate local response times for local statistics
A.aggregate_sub_tree    # combine local values with children's for statistics
                        #  of the whole subtree
A.value                 # values of statistics of subtree (including own)
Comm.send_to_self       # send a message to the own node with a delay
Comm.send_to_neighbors  # send to all specified neighbors
Comm.send_to_parent     # send to parent
find_parent             # among neighbors, choose first with minimum level as
                        #  new parent

procedure GAP( )
    DELAY = # fixed time window
    Neighbors := empty;
    ResponseTimes := empty;

    if v = root then
        level = 0
        parent = 0
    else
        level = INF
        parent = INF
    end if

    A.initiate()
    while true do
        read message;
        switch (message)
            case (ResponseTimeArriveMessage, responseTime)
                ResponseTimes.add(responseTime)
```

```
                # to keep only a set of responseTimes in cache
                Comm.send_to_self(DELAY, TimeOut(responseTime))
                A.aggregate_local(ResponseTimes)
                A.aggregate_sub_tree(Neighbors)
            case (UpdateVector, neighbor):
                Neighbors.add(neighbor)
                (newParent, newLevel) = find_new_parent(Neighbors)
                A.aggregate_sub_tree(Neighbors)
                if level != newLevel then
                    level = newLevel
                    parent = newParent
                    Comm.send_to_neighbors(Neighbors, level, parent, A.value())
                    continue
                end if
            # to keep only a set of responseTimes in cache
            case (TimeOut, responseTime):
                responses.remove(responseTime)
                A.aggregate_local_value(ResponseTimes)
                A.aggregate_sub_tree(Neighbors)
        end switch
        Comm.send_to_parent(Neighbors, A.value)
    end while
    end procedure
```

1.2 Implementation Details

A new base class was implemented which all of the tasks extend: `peersim.EP2300.vector.GAPNode`. In `peersim.EP2300.message` three new messages were introduced: `ResponseTimeArriveMessage`, `UpdateVector`, `TimeOut`. The first two roughly equal `LOCALVAR` and `UPDATE` from the original technical report (see [1]).

Each node stores all its information on it's neighbors as `NodeStateVector` in a SortMap named `neighborList`. Details on the nodes response times are stored in the ArrayList `requestList`. To focus only on the response times in the current time window, every response time is assigned a time out.

For the implementation with rate control, every message send to the parent will trigger a decrease of the msg budget. As soon as the msg budget is lower or equal to 0, no more messages are sent out. The message budget is reset via a new control `peersim.EP2300.control.ResetMsgBudget`.

1.3 Results

(i) time series of f(t) and f(t) for r = {0.2,0.4,0.8} and {R1} from the first 5 minutes

(ii) time series of f(t) and f(t) for r = {0.2,0.4,0.8} and {R1} after the first 5 minutes

(iii) trade-off plots for both {R1,R2} (and all rate options?? 0.1,0.2,0.4,0.8,1.6)

(iv) density plots for r = {0.2, 0.4, 0.8} and {R1,R2}

Task 2

Listing 2: inspired by [1]

```
messages:
(ResponseTimeArriveMessage, responseTime)
(UpdateVector, neighbor)
(TimeOut, responseTime)

objects and help functions:
A       # storing values for and performing aggregation
Comm    # communication messages
A.aggregate_local       # aggregate local response times for local statistics
A.aggregate_sub_tree    # combine local values with children's for statistics
                        #  of the whole subtree
A.value                 # values of statistics of subtree (including own)
Comm.send_to_self       # send a message to the own node with a delay
Comm.send_to_neighbors  # send to all specified neighbors
Comm.send_to_parent     # send to parent
find_parent             # among neighbors, choose first with minimum level as
                        #  new parent

procedure GAP( )
    DELAY = # fixed time window
    Neighbors := empty;
    ResponseTimes := empty;

    if v = root then
        level = 0
        parent = 0
    else
        level = INF
        parent = INF
    end if

    A.initiate()
    while true do
        read message;
        switch (message)
            case (ResponseTimeArriveMessage, responseTime)
                ResponseTimes.add(responseTime)
                # to keep only a set of responseTimes in cache
                Comm.send_to_self(DELAY, TimeOut(responseTime))
                A.aggregate_local(ResponseTimes)
                A.aggregate_sub_tree(Neighbors)
            case (UpdateVector, neighbor):
                Neighbors.add(neighbor)
                (newParent, newLevel) = find_new_parent(Neighbors)
                A.aggregate_sub_tree(Neighbors)
                if level != newLevel then
                    level = newLevel
                    parent = newParent
                    Comm.send_to_neighbors(Neighbors, level, parent, A.value())
                    continue
                end if
            # to keep only a set of responseTimes in cache
            case (TimeOut, responseTime):
                responses.remove(responseTime)
                A.aggregate_local_value(ResponseTimes)
                A.aggregate_sub_tree(Neighbors)
        end switch
        Comm.send_to_parent(Neighbors, A.value)
    end while
    end procedure
```

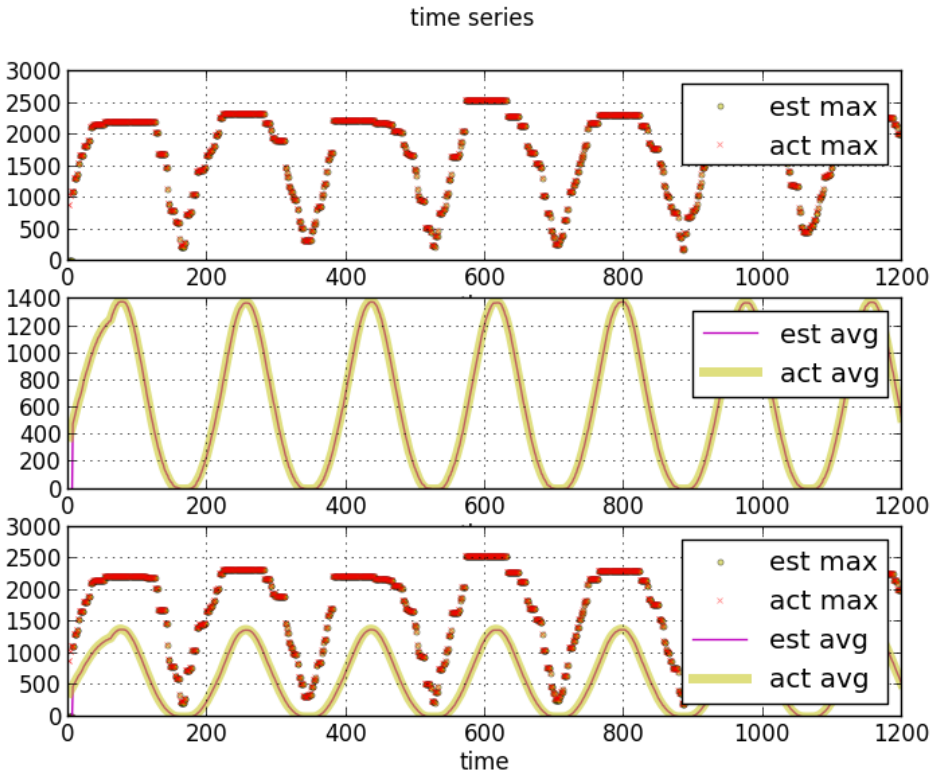

Abbildung 1: Time series plot

- compare performance for R_1, R_2
  - time series of f (t) and f(t) for r = { 0.2, 0.4, 0.8 } amd $\{R_1,\ R_2\}$ from the first 5 min
  - trade off plot for R_1, R_2
  - density plot for r = { 0.2, 0.4, 0.8 } and $\{R_1,\ R_2\}$

2 Task II

- pseudo code!
- implementation details
- compare performance for R_1, R_2
  - time series of f (t) and f(t) for r = { 0.2, 0.1, 0.05, 0.025, } amd $\{R_1,\ R_2\}$ from the first 5 min
  - trade off plot for R_1, R_2

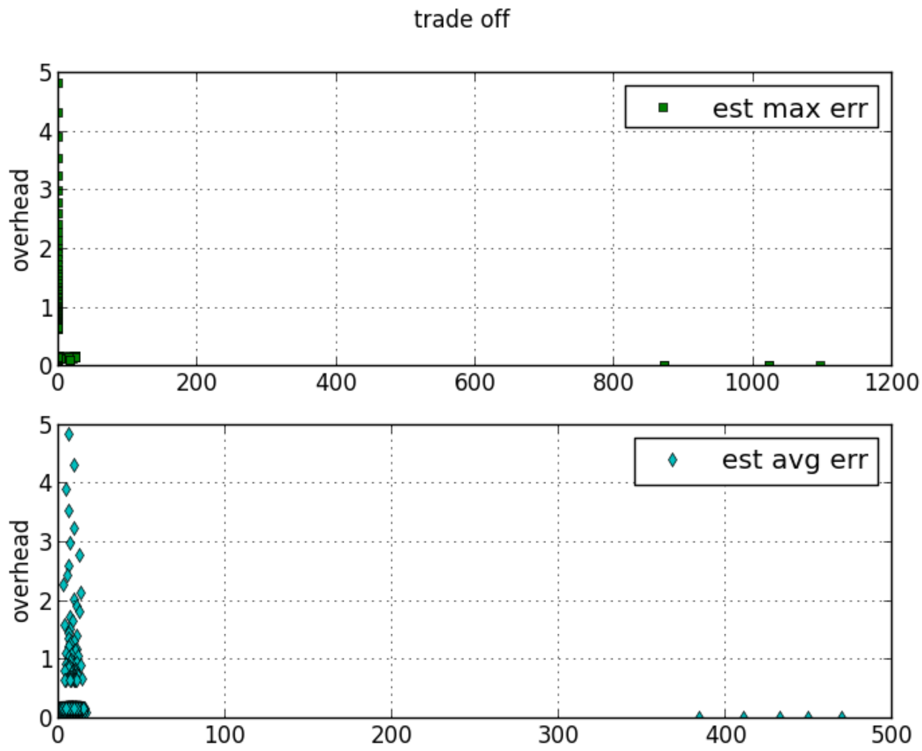

Abbildung 2: Trade off plot

- density plot for r = { 0.1, 0.05, 0.025 } and $\{R_1, R_2\}$

3 Task III

- pseudo code!
- implementation details
- compare performance for R_1, R_2
  - time series of f (t) and f(t) for r = { 0.2, 0.1, 0.05, 0.025, } amd $\{R_1, R_2\}$ from the first 5 min
  - trade off plot for R_1, R_2
  - density plot for r = { 0.1, 0.05, 0.025 } and $\{R_1, R_2\}$

4 Summary

- Compare TaskI, II, III

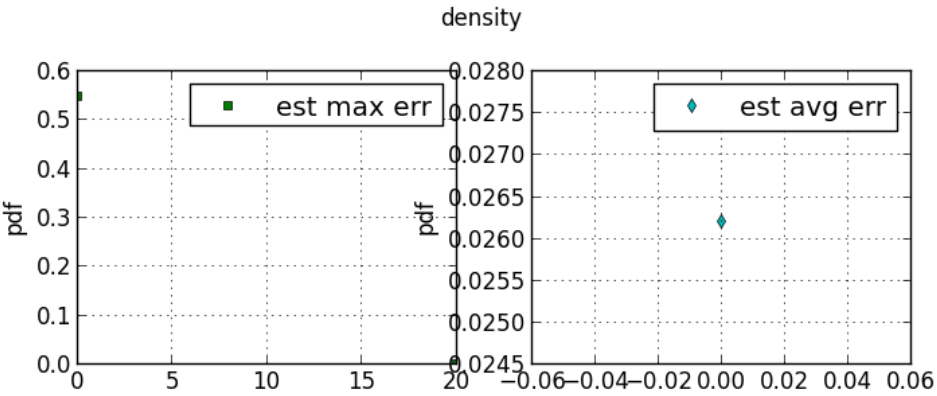

Abbildung 3: PDF plot

- Compare R_1, R_2 globally

Literatur

[1] R. Stadler, "Protocols for distributed management," 2012.