Appendix B

BUG’s C++ implementation for the simulation experiments

```cpp
#ifdef FLOWSCGHS4_H
#define FLOWSCGHS4_H

class FlowSchedulerGPS : public ActiveObject
{
private:
  FlowControlGPS4 * controls;
  // CustomerClass objects it schedules.
  unsigned int numControls;
  // Number of services, need it for accelerate the sorting.
  CustomerClass * cost;
  // Customer class representing computing cost.
  CustomerClass * signal;
  // Customer class receiving processing cost signals.
  Time period;
  // Computation period.
  Time timeManaged;
  // Period's fraction managed.
  Time nextActivation;
  // This is required because the instant when we know when the next activation will take
  // place and the instant when we can set our event at that time value are not the same.
  // And they are not the same due to the implementation of the grants adjusted signaling
  // mechanism.
  Time lastActivation;
  // This is required for the solution's implementation of the "Fully utilized regulated
  // ServiceStation detection" mechanism.
  bool signalGrantsAdjusted;
  // Flag to indicate that the current event is just for letting ServiceStations
  // to notice their grants have been adjusted.
  bool monitoring;
  bool enforcing;
  // These are the exclusive operation modes.
  bool conservative;
  // Use a conservative policy. XXX not used here.
public:
  FlowSchedulerGPS(int, double, bool = false, const char * = NULL);
```
// Default constructor. Give name, scheduling period and per-period manage capacity.
void addFlowControlGPS(CustomerClass *, CustomerClass *, double, double, int);
// Associates a Schedule to a CustomerClass object.
void setSchedule(CustomerClass *);
// Sets the customer class it hands customers to.
void setExtract(CustomerClass *);
// Sets the customer class it extracts customers from.
void service(Event event);
// Performs its service onto its current service if there is one.
void scheduling(Event event);
// Selects the next current Customer from the Service objects it serves.
void clear(void);
// Clear statistics.
void printStats(Time total); // Print statistics.
}
#endif // FLOWCHGSPS4.H

#include <stdio.h>
#include <stdlib.h>

#include "object.h"  
#include "namedobj.h" 
#include "logobj.h"  
#include "mytime.h" 
#include "event.h"  
#include "customer.h" 
#include "queueobj.h"  
#include "arrtimeo.h"  
#include "cstclasse.h"  
#include "actobje.h" 
#include "flowctrlgps4.h" 
#include "flowschgps4.h"


// A FlowScheduler is meant to indirectly regulate the utilization of some ServiceStation.
// In order to do this, a FlowScheduler manipulates the _vacancy_space_ of a CustomerClass
// that is the target of a _blockable_Transition that is part of the regulated ServiceStation.
// Observe that if the Transition is not blockable, the FlowScheduler cannot accomplish its
// objective. (It nearly produces a lot of dropping in the target CustomerClass.) Generally,
// the FlowScheduler's actions are based on information gathered from the two CustomerClass
// at either end of a blockable Transition.
//
// NOTE: After manipulating the vacancy space of all target CustomerClasses we need to immediately
// produce a system event so any blocked ServiceStation may resume servicing Customers, if
// appropriate. Otherwise, these ServiceStations will be idle up to the next event which may
// never occur.
//
// In this implementation, objects from the FlowControl class are the ones responsible for
// information gathering and vacancy space manipulation. Thus, a FlowScheduler object is
// associated to as many FlowControl objects as there are Transitions in the regulated
// ServiceStation. The FlowScheduler centralizes the information from its FlowControls and
// implements the scheduling policy. But the FlowScheduler never interacts with any
// CustomerClass objects.
//
// NOTE: possible problems may arise due to differences between parameter's units. For instance, rounding
// errors may appear if some parameters are measured with Customer size units and others as
// Customer counts.
//
// BUG is a periodic FlowScheduler that approximates the behavior of a PGPS. It is only an
// approximation because while the PGPS acts on every Customer arrival and departure, BUG only
// acts every T time units. Because in general T > L, the maximum Customer service time, and
// it could be that T > IAT, some Customer interarrival time, several PGPS events may take place
// during a BUG's activation period. In fact, during a system's busy period this is always true.
// Consequently, BUG acts reactively: that is, from a summary of what have happened at the regulated
// ServiceStation and Customer flows during the last T time units, BUG adjusts vacancies spaces at
// the target CustomerClasses so the regulated ServiceStation's behavior for a T period (from –T
// to T, where BUG's current time is 0) resembles that of a PGPS. To do this, BUG runs an emulated
// PGPS who offered load, resource allocation period, and computed use-grants are manipulated by
// the BUG following a closed-loop control-logic. This control logic's objective is to minimized,
// at every activation period, the difference between the behavior of the regulated ServiceStation
// and that of a real PGPS.
//
// NOTE: possible problems may arise due to correlations between the size of the activation period T and
// both the duration of a Customer service time L and size of the interarrival time IAT. For
// instance, rounding errors may appear if T/L != 0 or T*IAT != 0.
This is a work-conservative implementation in which any ServiceStation's idle time detected at
the emulated PGFS is given away for all customer flows to use. Besides, this implementation
initially sets all use-grants to a value proportional to a complete T period, so the
ServiceStation can get busy as soon as possible. This implementation has two evident
implications, one is good and the other is bad. On the good side, customers do not experience
any initial waiting. On the bad side, some ServiceStation-use unfairness may arise because some
customer flow may use more than its fair share. Consequently, BUG implements a counterbalancing
mechanism so during the next T time units the deprived customer flows recover and the depriver
ones get restrained. This mechanism is explained below. Also observe that unfairness can only
occur if the above situation happens during a ServiceStation 100% utilization period.
Consequently, BUG need to keep track of the utilization level of the regulated ServiceStation.
Let us now consider another, may be not so evident, bad implication of this implementation. Here,
the source CustomerClasses' occupation levels are not, in general, a measure of the new arrivals
for the next activation period, and thus BUG cannot use these levels as the input load for the
emulated PGFS. Instead, BUG needs to compute this input load from the running sum of arrived
bytes that each transitions' source CustomerClasses must keep. Finally, BUG here uses a "floor"
semantic when packetizing the byte use-shares given by the emulated PGFS and thus some mechanism
must be provided so no ServiceStation idle time is artificially produced.

NOTES:

Activation period:

BUG normally works periodically with period T, but the implementation of the solutions to the
"Fractional Customers" and the "Rounding off of byte shares" problems may produce variations
to the period length. Consequently, it is not possible to know in advance the time of
activation of future events. Thus, each time an activation event expires BUG computes and
stores a next-activation proposed time T into the future so if and after it learns that
no period length variations will be required it can set the event to that value. As an
implementation decision, the system's first event (the initial event) is taken as an activation
mark.

Cost:

We implement the following mechanism for modeling the possible extra cost associated with
the BUG algorithm. At every activation instant, the FlowScheduler sources and delivers, at
ServiceTime, a "cost" CustomerClass. The network should convert this
"cost" Customer into a "signal" Customer and deliver it to the signal CustomerClass. Because
the system moves Customers during service time, "signal" detection is best done at scheduling
time. When a "signal" arrives we take the scheduling actions.

When are the actions taken?

Because we are modeling the costs associated with BUG actions and we are implementing
this as a "cost" Customer being serve at some ServiceStation, BUG actions are actually
taken at T+dt, where dt depends on the kind of scheduling used in and the load presented
to the ServiceStation that serves the "cost" Customer.

Operation modes:

* Monitoring mode. It is the initial mode and BUG remains in it as long as the utilization
level of the regulated ServiceStation during the past activation period is below 100%. When
at this mode, BUG do not run the emulated PGFS and gives away to all customer flows
ServiceStation use grants proportional to T.
* Enforcement mode. BUG enters this mode when the utilization level of the regulated
ServiceStation during the past activation period is at or above 100%. (Utilization levels
above 100% may happen due to the implementation of the solution to the fractional Customers
problem.) When at this mode, BUG runs the emulated PGFS and allocates ServiceStation use
grants accordingly to the PGFS outputs and the unfairness counterbalancing mechanism. (See
Unfairness counterbalancing.) BUG will exit this mode, entering monitoring mode, when the
mentioned utilization level drops below 100%.

Fully utilized regulated ServiceStation detection:

This BUG implementation requires detecting when the regulated ServiceStation is fully
utilized. One way to do this is letting the FlowScheduler to ask its regulated ServiceStation
for its idle time at each activation period. If there is a difference between the current
value and the read previously then the ServiceStation WAS NOT fully utilized during the last
T time units. The problem with this approach is that several changes are required outside this
file. (We need to implement an association between ServiceStation objects and FlowScheduler
objects. Also, we need to implement the method for questioning the ServiceStation objects.
Moreover, we need to extend the configuration file's section on FlowSchedulers and its
corresponding parser.) Another way to do the detection is letting each FlowControl to ask
its destination CustomerClass for its Customer size count for arrived Customers at each
activation period and from all the answers to compute the utilization percentage. This is
computationally costlier but at this time requires no changes outside this file. Moreover,
because in general the length of an activation period varies (see Activation period), with
this approach BUG need to remember the time of last activation so the period length can be
computed. In any case, observe that if the regulated ServiceStation enters a 100% used period
at the middle of an activation period (after that unfairness may occur), the utilization
level measured for the period will be less than 100%. It seems this is something BUG cannot
solve.

(XXX If a get some time I MUST implement the first solution as this is the best thing to do.)
Unfairness counterbalancing:

There are two possible mechanisms. One manipulates the PGFS inputs and the other manipulates
the PGPS outputs. Either way, the mechanisms work differently whether BUG is at monitoring
or enforcing mode. (See Operation modes.) When in monitoring mode, if the regulated
ServiceStation was 100% used during the last activation period, for each Customer flow BUG
compares the flow's ServiceStation utilization with the nominal use share. BUG then manipulates
in some way either the PGPS input load for the flow or its output, depending if the nominal use
share is smaller (which indicates this flow is a deprriver) or greater (which indicates this
flow was deprived). When in enforcing mode, BUG constantly compares the running sum, from the
moment it enters into enforcing mode, of the above mentioned values for each Customer flow,
and adjusts either the PGPS input load for the flow or its output, accordingly.

* Manipulating PGPS inputs. In this case the PGPS input load of deprriver flows are reduced by the
  compared values' positive difference and the ones for deprived flows are raised by the
  corresponding difference. If arrivals at the deprrivers are not above nominal values, this
  mechanism will result in less use grants for the deprrivers and more use grants for the
  deprived ones. This means that although during a transient overload the use share will be
  fair, the overall unfairness will last until the deprriver's load returns to nominal values.

* Manipulating PGPS outputs. In this case the PGPS outputs—the use shares—of deprriver flows
  are reduced by the compared values' positive difference and the ones for deprived flows are
  raised by the corresponding difference, but only if the PGPS was also at 100% used during the
  current emulation run. One problem with this approach is that the time required for servicing
  through the use of these shares may be less than T and thus it can artificially produced some
  ServiceStation idle time. Happily, this problem is solved by the mechanism that deals with the
  rounding off of byte shares. (See Rounding off byte shares.)

Rounding off byte shares:
The rounding off of packet grants to the smallest integer from the byte shares given by the
emulated PGPS may produce a problem. If, for example, some byte share equals to 1.5 packets,
the overall share assigned to that flow will be almost half of what it should be. This problem
can be solved observing the following. On one hand, unsuited share bytes can only occurred
when the PGPS's utilization, and thus that of the regulated ServiceStation, is at 100%. On the
other hand, the time required for servicing the packet grants is smaller than T only by a
fraction of a packet serving time. Being so, we can let BUG reduce its activation period for
the next activation to T-dt, when dt will be the fractional packet serving time, without
increasing to much the overhead. Observe that at T-dt all granted Customers have just left the
regulated ServiceStation, and that during the last T-dt time units the regulated ServiceStation
was 100% utilized. Thus, the current occupation levels at the source CustomerClasses are what
the emulated PGPS is expecting.

Fractional Customers (an important consequence):
In general, BUG's activation period is unrelated to Customer's transmission time through the
regulated flows. Thus, if care is not taken when configuring the queuing network model, it
may happen that during monitoring mode (see Operation modes) BUG actions take place while
some Customer is being served by the regulated ServiceStation. (This cannot happen during
enforcing mode because at that mode the use grants computed by BUG are always served in as
much as T time units.) We name such a Customer a fractional Customer. Fractional Customers may
produce some errors. One is that the occupation level reported by a CustomerClass may be wrong:
it will include (at sources) or omit (at targets) the whole fractional Customer size, while it
should only include the part that has or has not been served. Moreover and as a consequence
of this, the utilization level of the regulated ServiceStation computed by BUG during
monitoring mode may not match the reality. In the worst case BUG will not be able to notice
when the regulated ServiceStation gets 100% used and thus will never enter enforcing mode.
A second error may result in a fractional Customer being dropped. This happens if BUG actions
result in a zero vacancy space for the fractional Customer's target CustomerClass. For solving
the first problem, the queuing network model MUST be configured so no fractional Customers
exist. To do this, the "cost" Customer can be made to be served by the regulated
ServiceStation after exiting the "cost" ServiceStation, before being deliver to BUG's "signal"
CustomerClass. For solving the second problem, the first problem has to be solved and it has
to be assured that the FlowScheduler is signaled for scheduling (when its actions are taken)
before the regulated ServiceStation is. Up to the current release, this can only be done
during system configuration by adding the FlowScheduler before the regulated ServiceStation to
the Chronos ActiveObjects list.

Sequence of events:
nt Service: activation detection and cost delivery.
nt+cost_time+queuing_time) Scheduling: signal detection and take scheduling actions.

// Default constructor.
FlowSchedulerGPS:FlowSchedulerGPS(
  int       _period,
  double    _managedCapacity,
  bool      _conservative,
  const char * _name)
  : ActiveObject(EVENT_SRVCC, _name),
  period(_period),
  lastActivation(0)
{
  if (_managedCapacity > 1.0 || _managedCapacity <= 0.0)
    error(FLWSCHEDED);
  timeManaged = period * _managedCapacity;
}
controls = NULL;
signal = cost = NULL;
umControls = 0;
conservative = _conservative;
signalGrantAdjusted = false;
 enforcing = !(monitoring = true);
}

// Add service

// Associates a service to a CustomerClass object.
void FlowSchedulerGPS::addFlowControlGPS(
    CustomerClass* _source,
    CustomerClass* _destination,
    double _cost,
    double _share,
    int _customerSize
) {
    FlowControlGPS* _fcNode;

    // Check that the CustomerClass is not served already.
    // If it is, signal an error.
    if (controls != NULL && controls->doHaveIt(_source))
        error(FLowsched1);
    // Spawn a new Schedule association and initialize the object.
    _fcNode = new FlowControlGPS(_source, _destination, _cost, _share, _customerSize, timeManaged);
    if (_fcNode != NULL)
        error(FLowsched2);
    // Add it to the list.
    controls = _fcNode->add(controls);
    ++numControls;
}

// Sets the customer class it hands customers to.
void FlowSchedulerGPS::setHandle(
    CustomerClass* _custClass
) {
    if (_custClass == NULL)
        error(FLowsched3);
    cost = _custClass;
}

// Sets the customer class it extracts customers from.
void FlowSchedulerGPS::setExtract(
    CustomerClass* _custClass
) {
    if (_custClass == NULL)
        error(FLowsched3);
    signal = _custClass;
}

// Service routine.

// Here we only spawn and hand a customer to the handle. It is expected that
// the system, in the near future, will drive this customer up to the extract
// customer class. Only then this object will do its main work at scheduling time.

void FlowSchedulerGPS::service(
    Event _event
) {
    static int _activationCount = 0;
    if (cost == NULL)
        error(FLowsched5);

    if (_event.qral() || _event == event && !signalGrantAdjusted) {
        Customer* _customer = new Customer();
        if (_customer == NULL)
            error(FLowsched4);
        cost->add(_customer, _event.getTime());
        // Here we clear our event and compute the tentative time for the next activation.
        event.set(NEVER);
        nextActivation = _event.getTime() + period;
    }
// Log.
// The work is drive by the following distributed algorithm:
// 0) Find if the regulated ServiceStation was fully utilized during the last T time units.
// 1) Let each FlowControl to compute its part of the inputs for the emulated PGFS.
// 2) Run the emulated PGFS by exchanging signals with all the FlowControls. State information
// is kept in part within the FlowScheduler and in part within each FlowControl. The flow
// scheduler keeps the period's remaining time and the percentage of active flows, while each
// FlowControl keeps its corresponding flow's pending work.
// 3) Let each FlowControl to adjust its corresponding CustomerClass vacancy space from the
// information it produced during the running of the emulated PGFS.

// The emulated PGFS works as follows:
// - While the period's remaining time is not over and there are FlowControls with positive
// pending work time (liable to be serviced) and one of this is smaller than the period's
// remaining time (that is, it will finish its pending work before the end of the period), do
// a) emulate the passing of time by signaling all liable FlowControls to do some work as a
// function of the smallest pending work time among all liable FlowControls and the percentage
// of active flows. At this point, the FlowControl with the smallest pending work
// time is finished.
// b) update the period's remaining time reducing it by the amount of time just passed and update
// the percentage of active flows reducing it by the share of the FlowControl that just
// finished. (For the following iterations it is no longer liable.)

// Notes:
// - Observe that grants are adjusted initially so no matter what the size
// of the original vacancy at the customer classes was, it is set to an
// "optimized" value (depending on the flag for conservative operation).
// One way to implement the while loop for the emulated PGFS uses a sorting list of FlowControls
// holding the finishing time of each.

typedef struct {
    FlowControlGPS4 * control;
    Time time;
} SortingListNode;

static int compare(
    const void *a
    , const void *b
) {
    if (((SortingListNode *)a)->time < ((SortingListNode *)b)->time)
        return -1;
    if (((SortingListNode *)a)->time > ((SortingListNode *)b)->time)
        return 1;
    return 0;
}

void FlowSchedulerGPS4::scheduling(
    Event _event
) {
    Customer * __signalArrived = NULL;
    if (signal == NULL)
        error(FLOWSCHED6);
    if (__signalArrived != signal->currentCustomer) {
        bool _firstEnforcing = false;
        bool _gpsVacation = false;
    }
bool __psFullyUsed = false;
unsigned int _i;
FlowControlGPS4 * _fcNode;
SortingListNode * __sortingList = new SortingListNode[numControls];
Time __remainingTime = timeManaged;

// Acknowledge the signal.
signal->remove(_signalArrived);
delete _signalArrived;
if (amlogging()) {
    char _data[OBJECT_LOG_LEN];
    sprintf(
        _data
        , "BUG processing at %s\n"
        , _event.getSite().timeToS()
    );
    writelog(_data);
}

// Set the operation mode.
// Find if the regulated ServiceStation was fully utilized since the last activation.
// In this implementation we poll each flow control for its utilization and sum them all.
// Then we compare this with the manageable time.
// XXX This is not the best thing to do. See notes at the intro.
// Observe that the time that have passed since the last activation may be longer or
// shorter than T units due to fractional Customers. In any case, the resulting period
// must be scaled to the manageable time for computing the utilization.
do {
    double _utilization;
    Time _timeManagedLastPeriod = (_event.getTime() - lastActivation) * (period / timeManaged);
    Time __totalUsedTimeLastPeriod();

    for (_fcNode = controls; _fcNode != NULL; _fcNode = _fcNode->getNext())
    // __totalUsedTimeLastPeriod += _fcNode->timeUsedLastPeriod();
    // For taking care of rounding errors we are using an hysteresis method for
    // changing mode. That is, once we are at enforcing mode we'll stay there even though
    // the utilization drops a little bit.
    if (enforcing)
        enforcing = _utilization >= 0.97;
    else
        enforcing = _utilization >= 1.00;
    // Inputs for the PGFS are computed differently at the beginning of the first
    // enforcing period in a row. Thus, we have to detect this case.
    __firstEnforcing = enforcing && monitoring;
    monitoring = !enforcing;
    while(0);

    // If in enforcing mode do whatever necessary to control bus usage.
    // Else do nothing.
    if (monitoring) {
        // Just adjust use grants to give away all manage time.
        Time __grant;
        if (amlogging()) {
            char _data[OBJECT_LOG_LEN];
            sprintf(_data, "BUG is in monitoring mode\n");
            writelog(_data);
            sprintf(_data, "Next period's grants (in time units):\n");
            writelog(_data);
        }
        for (_fcNode = controls; _fcNode != NULL; _fcNode = _fcNode->getNext()) {
            __grant = _fcNode->adjustGrantMonitoring(timeManaged);
            if (amlogging()) {
                char _data[OBJECT_LOG_LEN];
                sprintf(_data, "%t\n", __grant.timeToS());
                writelog(_data);
            }
        }
    }
} else {
    // Enforcing.
    // if (amlogging()) {
    //     char _data[OBJECT_LOG_LEN];
    //     sprintf(_data, "BUG is in enforcing mode\n");
    // }
}
writelog(_data);
    printf(_data, "Next period's grants (in time units):\n");
    writelog(_data);
}

// Compute inputs at each FlowControl and initialize the sorting list.
//
// _fcNode = controls;
for (i=0; i < numControls; ++i) {
    // Inputs for the PGFS are computed differently at the beginning of the first
    // enforcing period in a row.
    _fcNode->computeInputs(_firstEnforcing);
    _sortingList[i].control = _fcNode;
    _sortingList[i].time = _fcNode->pendingWorkTime(1.0);
    _fcNode = _fcNode->getNext();
}

// Run the emulated PGFS.
//
// First the sorting.
qsort((void *)_sortingList, (size_t)numControls, sizeof(SortingListNode), compare);
if (_sortingList[numControls - 1].time > time Managed) {
    // If the largest pending work (the one at the end of the sorted list) is zero we are
    // in a vacation period and there is no need to compute work progress.
    _gpaVacation = true;
    _remainingTime = Time(0);
    for (i=0; i < numControls; ++i)
        _sortingList[i].control->doSomeWork(time Managed, 1.0);
} else {
    // Some flows will end before the end of the period and thus we have to compute
    // work progress by steps, changing at each step the PGFS state.
    unsigned int _firstNode = 0;
    double _percentageActiveFlows = 1.0;
    do {
        // Fist do some work at the current PGFS state.
        for (i=_firstNode; i < numControls; ++i)
            _sortingList[i].control->doSomeWork(_sortingList[i].time,
                                                _percentageActiveFlows);
        // Second update PGFS state.
        _percentageActiveFlows -= _sortingList[_firstNode].control->getShare();
        _remainingTime -= _sortingList[_firstNode].time;
        ++_firstNode;
        // Third recalculate pending work with the new PGFS state.
        // There is no need for sorting again.
        for (i=_firstNode; i < numControls; ++i)
            _sortingList[i].time = _sortingList[i].control->pendingWorkTime(_percentageActiveFlows);
        // Loop while there are liable flows and remaining time.
        while (_firstNode < numControls 
               && _sortingList[_firstNode].time < _remainingTime)
            ;
    // Do remaining work, if there is some.
    if (_firstNode < numControls) {
        for (i=_firstNode; i < numControls; ++i)
            _sortingList[i].control->doSomeWork(_remainingTime,
                                          _percentageActiveFlows);
        _gpaFullyUsed = true;
        _remainingTime = Time(0);
    }
}
// Adjust use grants.
//

Time _grant;
Time _sumGrant(0);

for (_fcNode = controls; _fcNode != NULL; _fcNode = _fcNode->getNext()) {
    _grant = _fcNode->adjustGrantEnforcing(_remainingTime);
    _sumGrant += _grant;
    if (amLogging()) {
        char _data[OBJECT_LOG_LEN];
        sprintf(_data, "%s\n", _grant.timeToS());
        writeLog(_data);
    }
    if (gpFullyUsed) {
        char _data[OBJECT_LOG_LEN];
        sprintf(_data, "Remaining time: %s\n", _remainingTime.timeToS());
        writeLog(_data);
    }
    if (_opsFullyUsed) {
        // If the POPS was fully utilized set the next activation instant not
        // to BUG's nominal period but to the end of the busy period. The length of
        // the busy period is equal to the grants sum, measured in time units, plus
        // the period's time not managed by the BUG.
        nextActivation = _event.getTime() + _sumGrant + (period - timeManaged);
    }
    while(0);
}

// Update FlowControls' state.
// XXX For optimization purposes this could be done in the same loop for
// adjusting the grants.
for (_fcNode = controls; _fcNode != NULL; _fcNode = _fcNode->getNext())
    _fcNode->updateState(enforcing, _firstEnforcing, _opsFullyUsed);

// Send signal for ServiceStations to notice the change.
signalGrantsAdjusted = true;
_event.setTime(_event.getTime() + 1);
// Now that we have finished our work we set the last activation mark.
// This mark may be set at any time after computing the bus utilization,
// but here, with all the updating state work, seems more appropriate.
lastActivation = _event.getTime();

// Clear stats.
void FlowSchedulerGPS::clear(void) {
}

// Print statistics.
void FlowSchedulerGPS::printStats{
    Time _totalTime
} {
}

#ifdef FLOWCTRLGPS4
#define FLOWCTRLGPS4 H

// Implements a GPS flow control.
// The idea is that this object manipulates the destination customer class' quota
// so the customer flow from the source through some service station gets controlled.

class FlowControlGPS4 : public Object {
    private:
        // CONTROL'S PARAMETERS.
        CustomerClass * source;
        // Source customer class.
        CustomerClass * destination;
        // Destination customer class.
        FlowControlGPS4 * next;
        // Pointer to form a list.
        double cost;
        // Processing cost in time units per customer size units.
        double share;
        // Policy assigned resource share.
        unsigned int customerSize;
        // Customer size units per customer.
        unsigned int fairShareNominalBytes;
        // Fair share measured in Customers size units which MUST be assured under a 100% load.
        static double totalShare;
}
// Accumulates the share of all existant controls.

// CONTROL'S STATE VARIABLES:
unsigned long lastSourceArrivedBytes;
// Number of Customer size units received at the source CustomerClass up to the last period.
unsigned long lastSourceDroppedBytes;
// Number of Customer size units dropped by the source CustomerClass up to the last period.
unsigned long lastDestinationArrivedBytes;
// Number of Customer size units received at the destination CustomerClass up to the last period.
int virtualQueueLevelBytes;
// Number of Customer size units store at the PGPS virtual queue.
int sumDestinationArrivedBytes;
// Running sum of Customer size units received during a busy period.
int sumFairShareBytes;
// Running sum of the PGPS computed fair shares measured in Customer for a busy period.

// CONTROL'S INSTANTANEOUS VARIABLES:
int workloadBytes;
// Input for the PGPS measured in Customer size units.
int pendingWorkBytes;
// Pending work inside the PGPS measured in Customer size units.

public:
FlowControlGP54(CustomerClass *, CustomerClass *, double, double, int, Time);
// Default constructor.
FlowControlGP54 * add(FlowControlGP54 *);
// Add to the list.
FlowControlGP54 * getNext(void) { return next; };
// Get next.
double getShare(void) { return share; };
// Get share.
void computeInputs(bool);
// Initialize scheduling variables and return the deviation serving time. 
Time timeUsedLastPeriod();
// Compute amount of time the controlled ServiceStation invested during the last period
// for serving Customers of this flow.
Time pendingWorkTime(double);
// Calculate pending work time.
void doSomeWork(Time, double);
// Change state for reflecting that some work has been done.
bool doIHaveIt(CustomerClass *);
// Check if a given customer class is in the list.
Time adjustGrantEnforcing(Time);
Time adjustGrantMonitoring(Time);
// Adjust the grant.
void updateState(bool, bool, bool);
// Adjust state.
void updateTotalShare(double);
// Update total share.
};

#endif // FLOWCTRLGP54_H

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include "object.h"
#include "namedobj.h"
#include "logobj.h"
#include "mytime.h"
#include "customer.h"
#include "queupoli.h"
#include "srvtimep.h"
#include "custclas.h"
#include "flowctrlgps4.h"

// A flow control is part of a flow scheduler, which is ment to indirectly regulate the
// utilization of some ServiceStation. In order to do this, a flow scheduler, by means of
// its flow controls, manipulates the _vacancy_space_ of CustomerClasses that are targets
// of _blockable_ Transitions that are part of the regulated ServiceStation. Observe that if
// the Transition is not blockable, the flow scheduler cannot accomplish its objective. (It
// nearly produces alot of droppings in the target CustomerClass.) Generally, the flow
// scheduler's actions are based on information gathered, again by means of its flow controls,
// from the two CustomerClass at either end of the blockable Transition.

// This file implements the flow controls for the BUG (especializado PGPS) flow scheduler.
// Sum of existant controls' share.
double FlowControlGPS4::totalShare = 0;

// Default constructor.
FlowControlGPS4::FlowControlGPS4(
    CustomerClass * _source
,   CustomerClass * _destination
,   double    _cost
,   double    _share
,   int       _customerSize
,   Time      _period
)
:
Object()
{
    if (_source == NULL || _destination == NULL)
        error(FLOWCTRL);
    if (_cost * _customerSize) / (cost * (double)customerSize) < 1.0)
        error(FLOWCTRL);
    if ((totalShare == _share) > 1.0)
        error(FLOWCTRL);
    source = _source;
    destination = _destination;
    cost = _cost;
    share = _share;
    customerSize = _customerSize;
    lastSourceArrivedBytes = 0;
    lastSourceDroppedBytes = 0;
    lastDestinationArrivedBytes = 0;
    virtualQueueLevelBytes = 0;
    sumDestinationArrivedBytes = 0;
    sumFairShareBytes = 0;
    fairShareNominalBytes = (unsigned int) floor((period.timeToF() * _share) / _cost);
    destination->getQueueStickyVacancy();
    next = NULL;
}

// Check if a given customer class is in the list.
bool FlowControlGPS4::doHaveIt(
    CustomerClass * _custClass
)
{
    if (source == _custClass || destination == _custClass)
        return true;
    if (next == NULL)
        return false;
    return next->doHaveIt(_custClass);
}

// Add to the list.
FlowControlGPS4 *
FlowControlGPS4::add(
    FlowControlGPS4 * _head
)
{
    // If there is no head, thus I'm the head.
    if (_head == NULL)
        return this;
    else {
        if (_head->next != NULL)
            add(_head->next);
        else
            _head->next = this;
    }
    return _head;
}

Time
FlowControlGPS4::timeUsedLastPeriod(void) {
    return Time((destination->arrivedBytes() - lastDestinationArrivedBytes) * cost);
}

// Compute inputs.
// We are implementing the non-manipulated-inputs algorithm.
// Weather this is or is not the first time into enforcing do:
// - If it is, the work load is taken from the occupation level at the source.
// - If it isn't, the work load is taken from the arrivals to the source plus the occupation
// level at the PGPS virtual queue, which was set during the last activation period when
// updating the state.
void FlowControlGPS4::computeInputs(
    bool _firstEnforcing
) {
    if (_firstEnforcing)
        workloadBytes = source->queueBytesLevel();
    else {
        // Compute how many bytes arrived and stay at the source during the last period.
        int _periodSourceArrivedBytes = source->arrivedBytes() - lastSourceArrivedBytes;
        int _periodSourceDroppedPeriod = source->droppedBytes() - lastSourceDroppedBytes;
        int _periodSourceArrivedStayBytes = _periodSourceArrivedBytes - _periodSourceDroppedPeriod;
        workloadBytes = _periodSourceArrivedStayBytes + virtualQueueLevelBytes;
    }
    pendingWorkBytes = workloadBytes;
}

// Calculate pending work.
Time FlowControlGPS4::pendingWorkTime(
    double _scale
) {
    return Time(pendingWorkBytes * cost / (share / _scale));
}

// Change state for reflecting that some work has been done.
void FlowControlGPS4::doSomeWork(
    Time _time
    double _scale
) {
    pendingWorkBytes -= (int)ceil(_time.timeToF() * (share / _scale) / cost);
}

// Adjust grant when at enforcing mode.
// Returns the time required to serve the computed grant.
Time FlowControlGPS4::adjustGrantEnforcing(
    Time _sparedTime
) {
    int _fairShareBytes = workloadBytes - pendingWorkBytes;
    int _grantBytes = 0;
    int _grantPackets = 0;
    int _periodDestinationArrivedBytes = destination->arrivedBytes() - lastDestinationArrivedBytes;
    int _unfairnessBytes = _periodDestinationArrivedBytes + sumDestinationArrivedBytes - sumFairShareBytes;
    if (workloadBytes < pendingWorkBytes)
        error(FLOWCTRL);
    // Compute a grant after the PGPS outcome and the unfairness level.
    // XXX May there be a problem when computing the unfairness level due to rounding off? Observe
    // that the PGPS computed fair share is not in general a integer multiple of the Customer
    // size while the bytes arrived at the destination is.
    //
    if (_unfairnessBytes > _fairShareBytes)
        // If the unfairness so far is greater than the fair share for the next period,
        // then choke the flow.
        _grantBytes = 0;
    else
        _grantBytes = _fairShareBytes - _unfairnessBytes;
    // In any case, add the spare time.
    _grantBytes += (int)ceil(_sparedTime.timeToF() / cost);
    // XXX What if the total is greater than what is available in the managedTime?
    // Within this simulator there is no problem as CustomerClasses verify that
    // the new queue vacancy never exceed the initial value given during configuration.
    // The configurator is responsible to give a proper initial queue size then.
    //
    // Packetize grant.
    _grantPackets = (int)floor((double)_grantBytes / (double)customerSize);
    // If the flow IS NOT being choked then we assure that at least one packet should go.
    if (_grantPackets == 0 && _unfairnessBytes < _fairShareBytes)
        _grantPackets = 1;
    //
// Adjusting.
//
// _grantPackets = destination->adjustQueueVacancyCustomers(_grantPackets);
//
// Finish.
//
// return Time(_grantPackets * customerSize * cost);
}

// Adjust grant when at monitoring mode.
// Returns the time required to serve the computed grant.
//
// Time
// FlowControlGPS::adjustGrantMonitoring(
//     _spareTime
// )
// {
//     _grantBytes = 0;
//     _grantPackets = 0;

// // Compute use grant.
// // Just give away the spare time.
// _grantBytes = (int)cell(_spareTime.timeToF()) / cost;
// // Packetize grant.
// _grantPackets = (int)floor((double)_grantBytes / (double)customerSize);
// // Adjusting.
// // destination->adjustQueueVacancyCustomers(_grantPackets);
// // Finish.
// // return Time(_grantPackets * customerSize * cost);

// Update the state.
//
// void
// FlowControlGPS::updateState(
//     bool _enforcing
// , bool _firstEnforcing
// , bool _gpsFullyUsed
// )
// {
// _fairShareBytes = workLoadBytes - pendingWorkBytes;
// _periodDestinationArrivedBytes = destination->arrivedBytes() - lastDestinationArrivedBytes;

// if (_enforcing) {
// // Update running sums.
// sumDestinationArrivedBytes += _periodDestinationArrivedBytes;
// if (_firstEnforcing && !_gpsFullyUsed)
// sumFairShareBytes += _fairShareBytes;
// else
// sumFairShareBytes += _fairShareBytes;
// // In enforcing mode the PGPS is being ran and we have to update PGPS's virtual queue.
// // For the virtual queue we cannot directly use the PGPS pending work due to the rounding
// // off of byte grants. The formula below gives a better number, which in general is larger.
// virtualQueueLevelBytes = workLoadBytes - _fairShareBytes / customerSize * customerSize;
// }
// else {
// // In monitoring mode we reset the state.
// sumDestinationArrivedBytes = 0;
// sumFairShareBytes = _fairShareBytes;
// virtualQueueLevelBytes = 0;
// }
// // This MUST always be done.
// lastSourceArrivedBytes = source->arrivedBytes();
// lastSourceDroppedBytes = source->droppedBytes();
// lastDestinationArrivedBytes = destination->arrivedBytes();
// }