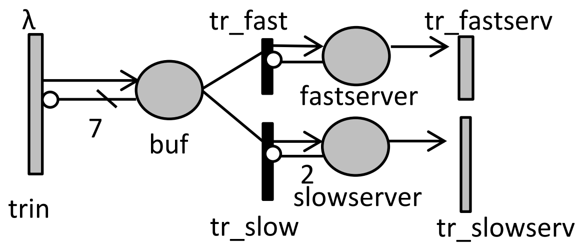
## CS 5214 Fall 2023

## Homework #3 Solution

### Problem #1

The SPN model for M/M/3/10 with a fast server and two slow servers is shown below:




**SPNP code:**

```
# include      "user.h"


       double lambda;

       double muf;

       double mus;

       int bufMax;

       int slowServerMax;

       int method;


parameters() {

       lambda = 15;

       muf=5;

       mus=3;

       bufMax=7;

       slowServerMax=2;

}


rate_type rate_mus()

{
```

```
    return mark("slowserver")*mus;
}


net() {
    place("buf");
    place("fastserver");
    place("slowserver");

    trans("tr_in");
    trans("tr_fast");
    trans("tr_slow");
    trans("tr_fastserv");
    trans("tr_slowserv");

    rateval("tr_in",lambda);
    probval("tr_fast",1.0);
    priority("tr_fast",2);
    probval("tr_slow",1.0);
    priority("tr_slow",1);
    rateval("tr_fastserv",muf);
    ratefun("tr_slowserv",rate_mus);

    oarc("tr_in","buf");
    mharc("tr_in","buf",bufMax);
    iarc("tr_fast","buf");
    iarc("tr_slow","buf");
    oarc("tr_fast","fastserver");
    harc("tr_fast","fastserver");
```

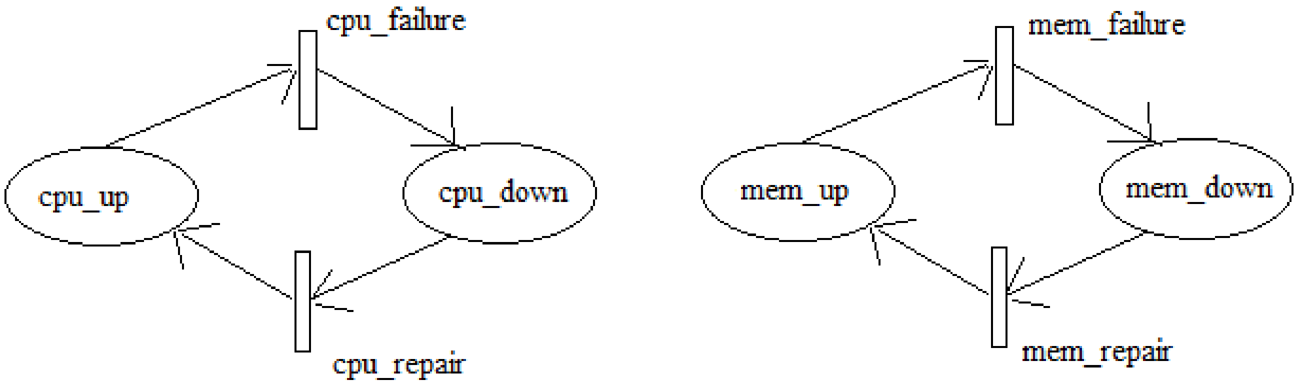
```
        iarc("tr_fastserv","fastserver");

        oarc("tr_slow","slowserver");

        mharc("tr_slow","slowserver",slowServerMax);

        iarc("tr_slowserv","slowserver");

}


assert() {

        if ( mark("buf") > bufMax )

                return(RES_ERROR);

        else

                return(RES_NOERR);

}

ac_init() {

        /* pr_net_info(); */

}

ac_reach() {

        /* pr_rg_info(); */

}

reward_type populationQueue() { return(mark("buf")); }
reward_type tput() { return(rate("tr_fastserv") + rate("tr_slowserv")); }
reward_type probrej() { if ( mark("buf") == bufMax ) return(1.0);
                                        else return(0.0); }

ac_final() {

printf("1. Customer Turned Away Probability = %f\n",expected(probrej));
printf("2. Average Number of Customers Waiting In The System = %f\n",expected(populationQueue));
printf("3. Response Time per client for those served by the fast server
= %f\n",expected(populationQueue)/expected(tput)+1/(double)muf);
printf("4. Response Time per client for those served by a slow server
= %f\n",expected(populationQueue)/expected(tput)+1/(double)mus);
printf("5. Throughput = %f\n", expected(tput));
}
```

**Output:**

```
SPNP Version 4.0
The analysis is starting.
The reachability graph contains:
    13 tangible markings
    17 vanishing markings
    50 arcs
After elimination of redundant arcs:
    # of remaining arcs:        50
After the elimination of vanishing markings:
    # of remaining arcs:        26
Solving the Markov chain...
...Markov chain solved
Reading the reachability graph info ...
1. Customer Turned Away Probability = 0.280006
2. Average Number of Customers Waiting In The System = 4.791933
3. Response Time per client for those served by the fast server = 0.643701
4. Response Time per client for those served by a slow server = 0.777034
5. Throughput = 10.799892
End of execution.
```

**Problem #2**

SPN Model




**SPNP source code**

```
 /**********************************
  * Hw#2, problem #2(b) using SPNP. *
**********************************/
#include<stdio.h>
#include "user.h"

const double lambdaP = 1.0/1000.0;
const double lambdaM = 1.0/1200.0;
const double muP = 1.0/4.0;
const double muM = 1.0/2.0;

parameters()
{
    iopt(IOP_METHOD, VAL_TSUNIF);
}

assert()
{
}

ac_init()
{
    pr_net_info();
}

ac_reach()
{
    fprintf(stderr,"\nThe reliability graph has been generated.\n");
    pr_rg_info();
}

rate_type cpu_repair_rate()
{
    return muP;
}
```

```
rate_type mem_repair_rate()
{
    return muM;
}

rate_type cpu_failure_rate()
{
    return (mark("cpu_up")*lambdaP);
}

rate_type mem_failure_rate()
{
    return (mark("mem_up")*lambdaM);
}

enabling_type enab_cpu_repair()
{
    /* allow cpu repair if the system is still alive */
    if(mark("cpu_up")>0 && mark("mem_up")>0)
       return 1;
    else
        return 0;
}

enabling_type enab_mem_repair()
{
/* allow mem repair if the system is still alive and there is no failed
   cpu because cpu repair has a high priority over mem repair*/

    if(mark("cpu_down")==0 && mark("cpu_up")>0 && mark("mem_up")>0)
           return 1;
    else
           return 0;
}


enabling_type enab_cpu_failure()
{
    /* allow cpu failure if the system is still alive */
    if(mark("cpu_up")>0 && mark("mem_up")>0)
       return 1;
    else
        return 0;
}

enabling_type enab_mem_failure()
{
/* allow mem failure if the system is still alive */
    if(mark("cpu_up")>0 && mark("mem_up")>0)
           return 1;
    else
           return 0;
}

net()
{
```

```
  /********************************
   * places/transitions/arcs for cpu *
   ********************************/
    place("cpu_up");
    init("cpu_up",2);
    place("cpu_down");

    trans("cpu_failure");
    ratefun("cpu_failure",cpu_failure_rate);
    guard("cpu_failure",enab_cpu_failure);

    trans("cpu_repair");
    ratefun("cpu_repair",cpu_repair_rate);
    guard("cpu_repair",enab_cpu_repair);

    iarc("cpu_failure","cpu_up");
    iarc("cpu_repair","cpu_down");
    oarc("cpu_failure","cpu_down");
    oarc("cpu_repair","cpu_up");

  /***********************************
     * places/transitions/arcs for memory *
   ***********************************/
    place("mem_up");
    init("mem_up",2);
    place("mem_down");

    trans("mem_failure");
    ratefun("mem_failure",mem_failure_rate);
    guard("mem_failure",enab_mem_failure);

    trans("mem_repair");
    ratefun("mem_repair",mem_repair_rate);
    guard("mem_repair",enab_mem_repair);

    iarc("mem_failure","mem_up");
    iarc("mem_repair","mem_down");
    oarc("mem_failure","mem_down");
    oarc("mem_repair","mem_up");
}

reward_type reliability()
{
    // The system is functioning if at least one CPU and memory

    if(mark("cpu_up")>0 && mark("mem_up")>0)
         return 1;
    else
         return 0;
}

ac_final()
{
    time_value(500.0);
    pr_expected("Reliability at time 500hr is", reliability);
}
```

**Output**

NET:

=================================================================================

places: 4

immediate transitions: 0

timed transitions: 4

constant input arcs: 4

constant output arcs: 4

constant inhibitor arcs: 0

variable input arcs: 0

variable output arcs: 0

variable inhibitor arcs: 0

=================================================================================

RG:

=================================================================================

tangible markings: 8 (4 absorbing)

vanishing markings: 0

marking-to-marking transitions: 11

=================================================================================

=================================================================================

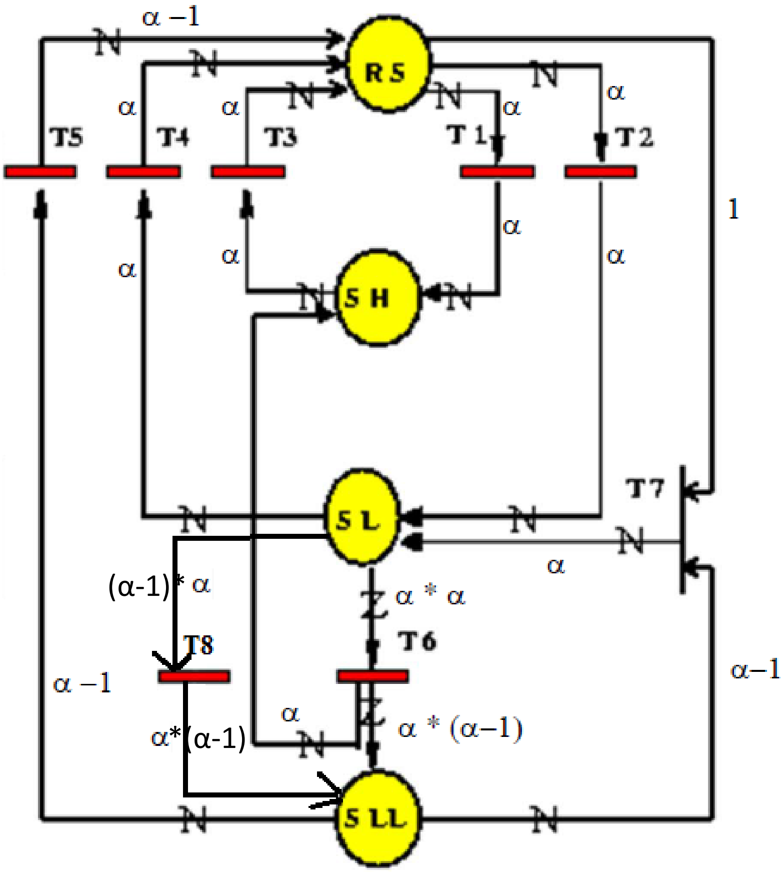
TIME :   500.000000000000

=================================================================================

EXPECTED: Reliability at time 500hr is = 0.994683960907

**Problem #3**

SPN Model




SPNP code

```
/***********************************
 * Solving Hw#2, problem #3 using SPNP.     *
 ***********************************/

#include<stdio.h>
#include "user.h"

const int alpha = 2;
const int numOfTokens = 6; // #token in RS in the beginning
const double lambdaH = 3.0;
const double lambdaL = 4.0;
const double muH = 3.0;
const double muL = 4.0;

parameters()
{
}

assert()
{
}
```

```
ac_init()
{
    pr_net_info();
}

ac_reach()
{
    fprintf(stderr,"\nThe reliability graph has been generated.\n");
    pr_rg_info();
}

rate_type t3()
{
    return ((double)mark("SH")/alpha * muH);
}

rate_type t4()
{
    return ((double)mark("SL")/alpha * muL);
}

rate_type t5()
{
    return mark("SLL") * muL;
}

/****************************************************************
 * This function returns true when there is no available slot. *
 ****************************************************************/
enabling_type isZeroTokenInRS()
{
    if (mark("RS") == 0)
        return 1;
    else
      return 0;
}

net()
{
    place("RS");
    init("RS", numOfTokens);
    place("SH");
    place("SL");
    place("SLL");

    trans("T1");
    rateval("T1", lambdaH);

    trans("T2");
    rateval("T2", lambdaL);

    trans("T3");
    ratefun("T3", t3);

    trans("T4");
    ratefun("T4", t4);
```

```
    trans("T5");
    ratefun("T5", t5);

/*************************************************************************************
 * T6 is designed for policy (ii).                                                   *
 * The incoming high priority client can lower the QoS                               *
 *   of two low-priority, high QoS clients.                                          *
 *************************************************************************************/
    trans("T6");
    rateval("T6", lambdaH);
    guard("T6", isZeroTokenInRS);

    trans("T7");
    priority("T7", 1);
    probval("T7", 1.0);

/************************************************************************************
 * T8 is added and designed for policy (iii).                                       *
 * The incoming low priority can lower the Qos of one low-priority, high QoS client. *
 ************************************************************************************/
    trans("T8");
    rateval("T8", lambdaL);
    guard("T8", isZeroTokenInRS);

    // arcs definitions

    miarc("T1", "RS", alpha);
    miarc("T2", "RS", alpha);
    miarc("T3", "SH", alpha);
    miarc("T4", "SL", alpha);
    miarc("T5", "SLL", alpha-1);
    miarc("T6", "SL", alpha*alpha);
    iarc("T7", "RS");
    miarc("T7", "SLL", alpha-1);
    miarc("T8", "SL", (alpha-1)*alpha);

    moarc("T1", "SH", alpha);
    moarc("T2", "SL", alpha);
    moarc("T3", "RS", alpha);
    moarc("T4", "RS", alpha);
    moarc("T5", "RS", alpha-1);
    moarc("T6", "SLL", alpha*(alpha-1));
    moarc("T6", "SH", alpha);
    moarc("T7", "SL", alpha);
    moarc("T8", "SLL", alpha*(alpha-1));
}
reward_type RejectionProbabilityHighPriorityClients()
{
// return 1 if there is no slot in the middle partition and T6 is not enabled
    return (mark("RS")==0 && !enabled("T6"));
}

reward_type numOfHighPriorityClients()
{
    return (mark("SH")/alpha);
}
```

```
reward_type ThroughputHighPriorityClients ()
{
    // the throughput of high priority clients.
    return rate("T3");
}

reward_type ThroughputLowPriorityLowQoSClients ()
{
    // the throughput of low-priority, low QoS clients.
    return rate("T5");
}

ac_final()
{
pr_expected("(a) The rejection probability of high-priority clients is ",
RejectionProbabilityHighPriorityClients);

pr_expected("(b) The average number of high-priority clients is ",
numOfHighPriorityClients);
pr_expected("(c) The throughput of high-priority clients is ",
ThroughputHighPriorityClients);
pr_expected("(d) The throughput of low-priority, low-QoS clients is ",
ThroughputLowPriorityLowQoSClients);
}
```

**output**

NET:

============================================================================

places: 4

immediate transitions: 1

timed transitions: 7

constant input arcs: 9

constant output arcs: 9

constant inhibitor arcs: 0

variable input arcs: 0

variable output arcs: 0

variable inhibitor arcs: 0

============================================================================

RG:

============================================================================

tangible markings: 16

vanishing markings: 9

marking-to-marking transitions: 54

================================================================================

EXPECTED: (a) The rejection probability of high-priority clients is  = 0.165820987343

EXPECTED: (b) The average number of high-priority clients is  = 0.834178917955

EXPECTED: (c) The throughput of high-priority clients is  = 2.50253675386

EXPECTED: (d) The throughput of low-priority, low-QoS clients is  = 0.657601089767