Determining The Requirement Of Medical Resources To Provide Timely Access To Health Care In Remote Areas

Project report for the evaluation of innovative work for MTE component

3rd semester, 2020-21

M&S SE-207

Submitted to: Prof. Rahul Chandra

Kshitiz Goel (2K19/SE/064)

Mayank Mittal (2K19/SE/071)

B.Tech Software Engineering

Department of Computer Science

Delhi Technological University

Bawana Rd, Shahbad Daulatpur Village, Rohini, Delhi, 110042

INDEX

ACKNOWLEDGEMENT

We would like to express our gratitude to all those who gave us the possibility to complete this project. We want to thank the Department of Software Engineering in the college for giving us permission to develop a project in this instance. The development of this project was only possible by the support, encouragement, cooperation of our teachers and we have got full support from our teachers .

We would also like to express heartfelt thanks to the people who are directly and indirectly part of this project. We have to thank our subject teacher Prof. Rahul Chandra for his help by providing the essential suggestions, help and the solution for the problem during project development.

ABSTRACT

Many regions are isolated and inaccessible due to topographical, developmental, and economical factors. Access to basic amenities like health care becomes a difficult task in such regions leading to deaths that could have been averted in settings with strong health systems.

A possible solution is to provide access to localized health services eliminating the requirement of commute and reducing delays in medical attention. The simulation aims to determine the number of beds and ambulances required to provide timely access to health care in a remote area minimizing the deaths caused due to such delays.

Model

The model is a Discrete Event Simulation (DES) model implemented via two consecutive queueing channels following the priority queue discipline:

- Patients (customers) and ambulances (servers)
- Patients (customers) and beds (servers)

The model draws patterns between certain parameters describing the remote area and the number of beds and ambulances required to ensure a timely response to health issues.

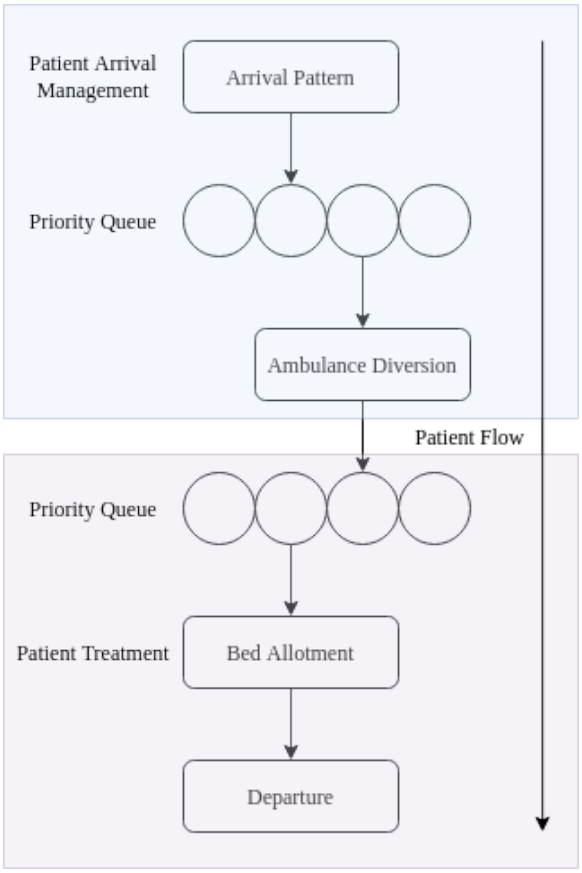


Fig. Flow of patients through the system

Our Approach

Requirements:

- A general-purpose computer for computational resources.
- Java development kit for programming and running the simulation model

Methodology:

General Assumptions

A remote area has a pre-defined population of which certain people become sick at regular intervals and require medical attention. The severity of sickness determines the level of priority given to the patient, the time for which a person can survive without medical attention, and the time needed for patients to recover. Delay in providing medical attention results in increased severity and eventually death.

Queue Behaviour Used: Reneging

If a person is suffering from such a severity level that if she waits for an ambulance for more time, then she will die before her chance comes. Henceforth, the person will be removed from the queue.

Queuing Channels

First Queuing Channel :

Ambulance (Servers) and Patients (Customers)

The number of people who become sick and the severity of their sickness is determined by relevant probability distribution functions and random number generators. Patients depending on priority are picked by ambulance. A random service time variable is considered to account for the distance of patients from the hospital. Failure in providing timely service results in death and increases the number of ambulances by one and resets simulation.

Implementation

- **Entities**
  - Patients (Customers)
  - Ambulance (Server)
- **System State**
  - Priority queue of patients
  - List of ambulances
- **Statistical counters**
  - Number of delays
  - Total Delay
  - The area under Q(t)
  - The area under B(t)

- **Time Profile**
  - Time of the last event
  - Current time
  - Next arrival time
  - Scheduled Departures
- **Routines**
  - Main Program
  - Initialization Routine
  - Timing Routine
  - Arrival Event Routine
  - Departure Event Routine
  - Library routine for generating uniform distribution random number.
  - Library routine for generating exponential distribution random number
  - Report Generator

In a Nutshell:

Till now, the simulation can determine the number of ambulances required in the remote area and generate the measures of performance according to the following parameters:

1. The maximum inter-arrival time between the patients
2. The radius of the remote area.
3. Maximum severity up to which injury/sickness is non-fatal (in terms of recovery time).
4. Maximum deaths are due to delay before an ambulance is added to service.

5. Ambulance's average speed as per infrastructural/topological conditions.
6. Rate parameter of the exponential probability distribution determining the initial severity

Result:

A sample result is shown which will show the output on specific parameters.

Inputs:

Parameter	Value
The maximum inter-arrival time between the patients	0.5 hours
Maximum severity up to which injury/sickness is non-fatal (in terms of recovery time)	30 km
Maximum deaths due to delay before an ambulance is added to service	10 hours
Ambulance's average speed as per infrastructural/topological conditions	30 km/hr
Rate parameter of the exponential probability distribution determining the initial severity	4

Output:

Parameter	Value
Number of Ambulances	4.3680
Average Delay	1.3433
Average Queue Length	5.3568
Percentage Utilization	90.7816

Second Queuing Channel :

Beds (Servers) and Patients (Customers)

Patients reach the hospital by ambulance and depending on priority are allocated beds. A service time variable is considered to account for the time of recovery. Failure in providing timely service results in death and increases the number of beds by one and resets simulation.

Implementation

- **Entities**
  - Patients (Customers)
  - Beds (Server)
- **System State**
  - Priority Queue of patients
  - Number of Beds in hospitals
- **Statistical Counters**
  - Number of Delays
  - Total Delay
  - The area under Q(t)
  - The area under B(t)
- **Time Profile**
  - Time of the last event
  - Current time
  - Next arrival time
  - Scheduled departures

- **Routines**
  - Main Program
  - Initialization Routine
  - Timing Routine
  - Arrival Event Routine
  - Departure Event Routine
  - Report Generator

In a Nutshell:

Now, we have completed the second simulation, wherein the Patients are Customers and Beds are the server.

We can now not only estimate the quantity of the beds required if the current disease parameters in suburbs/rural areas persist but also generate the relevant measures of performance.

Result:

Departure times and the severity of customers that departed were recorded and acted as inputs for the second queuing channel. Instead of random numbers, these values were used to calculate the required data.

Output:

Parameter	Value
Number of Beds	23.5680
Average Delay	0.1198
Average Queue Length	0.4890
Percentage Utilization	62.0645

Applications

Question:

In a rural town with a constant population density, of town radius: 30km is exposed to an infectious viral disease causing a pandemic.

There is only one Hospital located in the middle of the town. Now, the hospital is responsible for providing essential medical facilities like Ambulance and Beds.

The govt. Induces a no. of policies each month to control this outbreak. Consequently, the surge in no. of cases decreases, and the measure of this decrease is given by the inter-arrival time of the patients coming to the hospitals.

"More the inter-arrival time, less no. of cases reported in 24 hours."

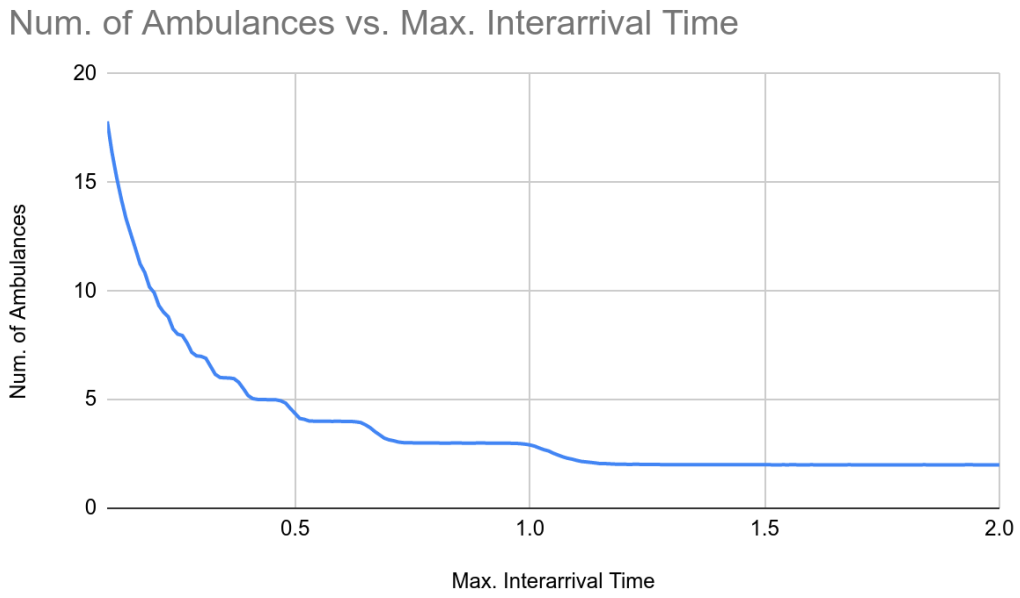
Simulate a model that will estimate the number of beds and ambulances required to minimize the deaths in that town for inter-arrival time ranging from 0.1 hours to 2 hours.

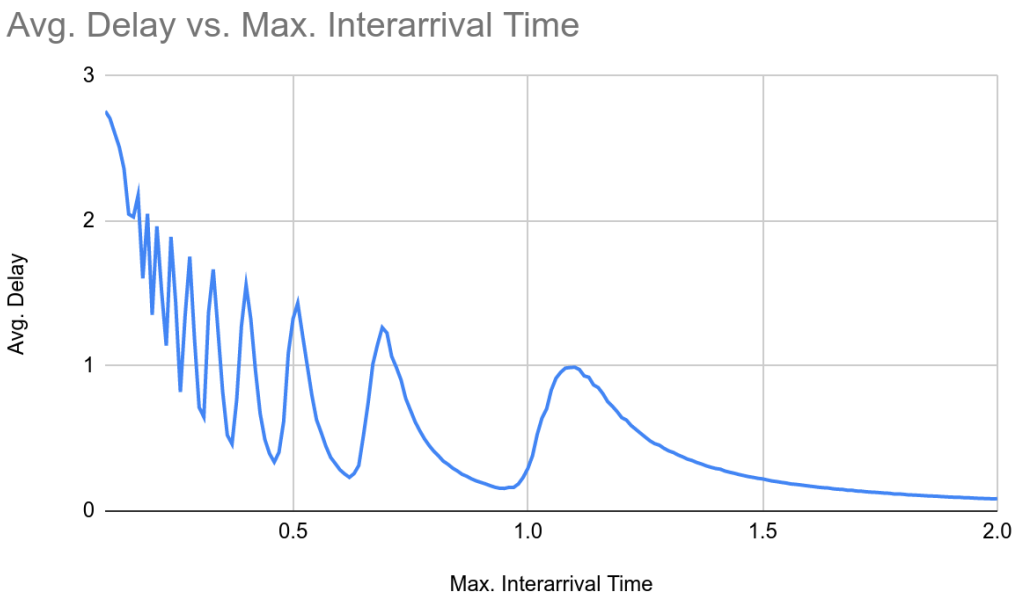
Solution :

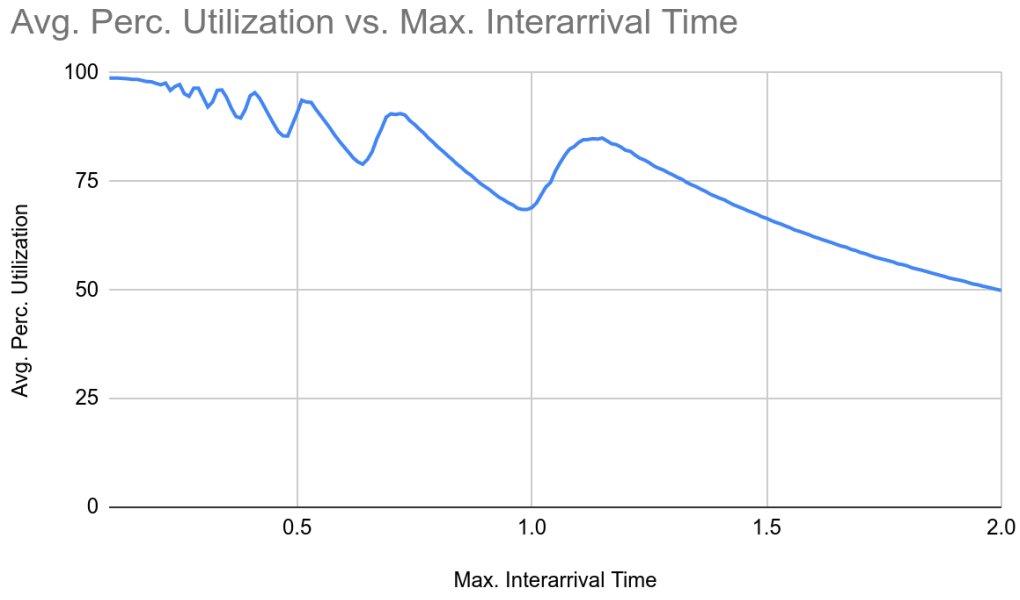
A program was made to determine the number of ambulances and the number of beds required to minimize the no. of deaths as much as possible.

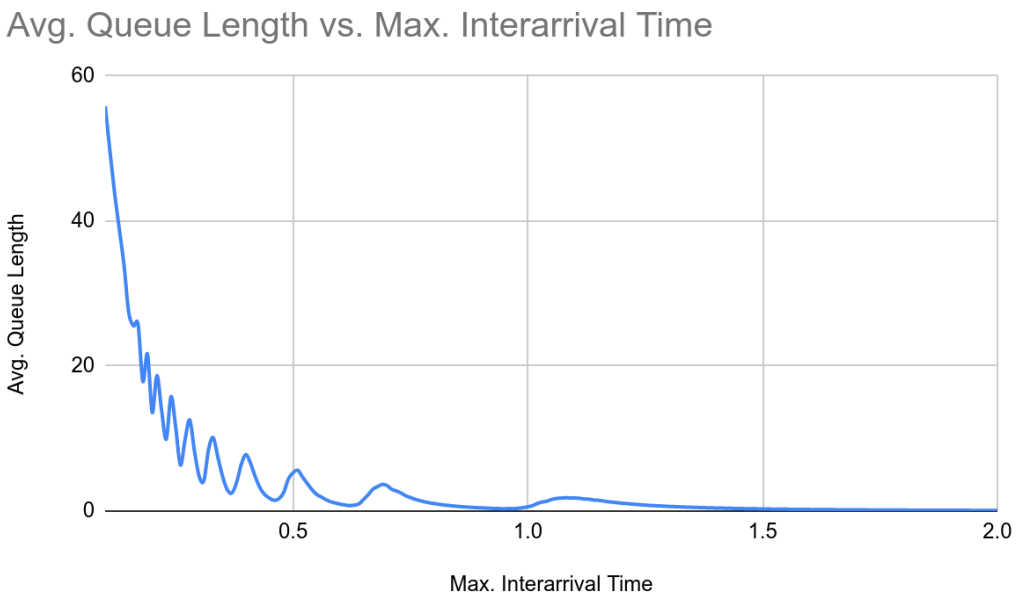
The following results were obtained:

Channel 1 (Patients and Ambulances):

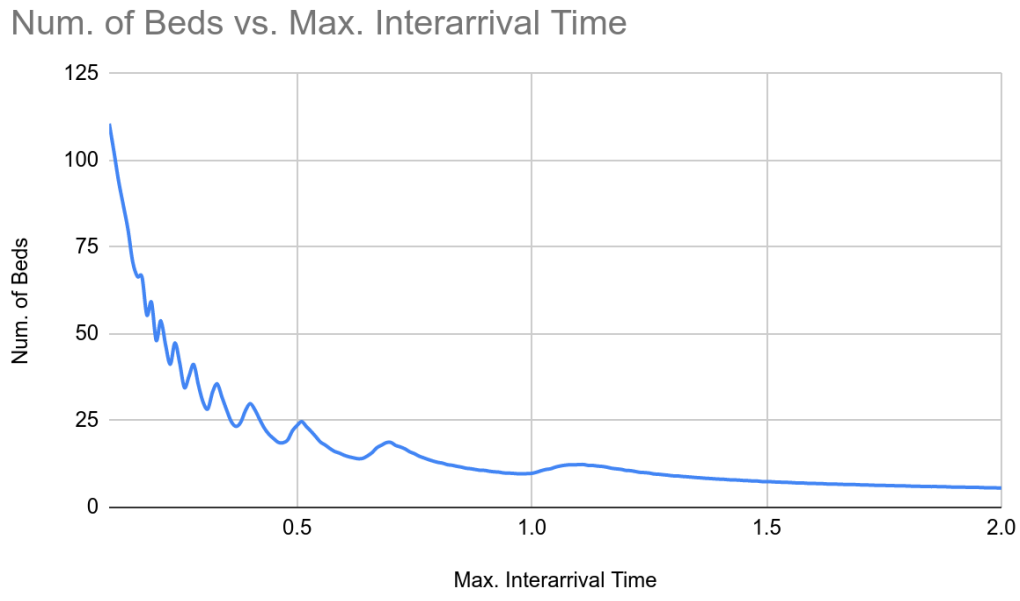


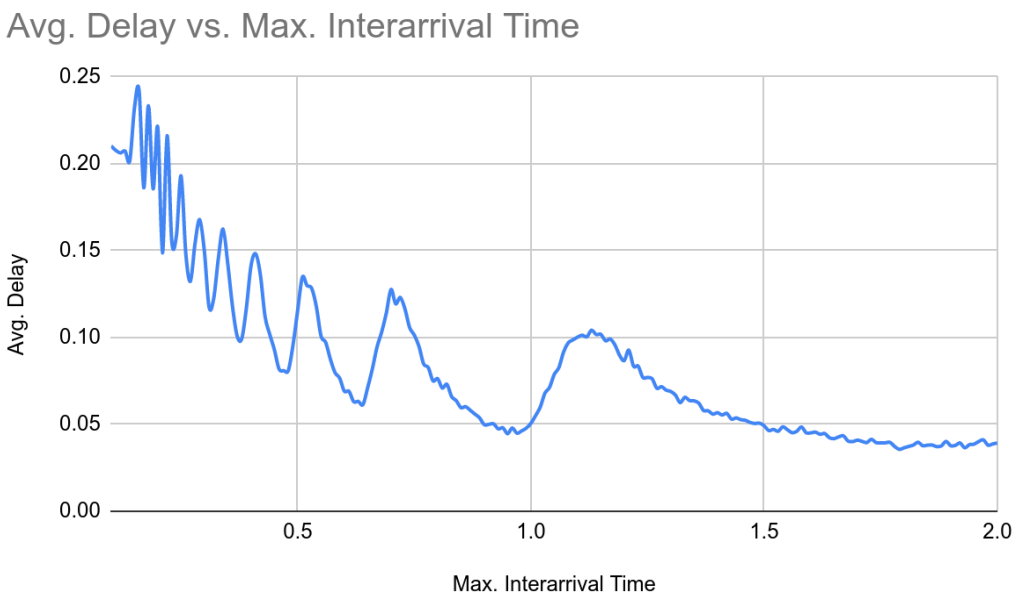

Avg. Delay vs. Max. Interarrival Time
Avg. Delay
3
2
1
0
0.5
1.0
1.5
2.0
Max. Interarrival Time

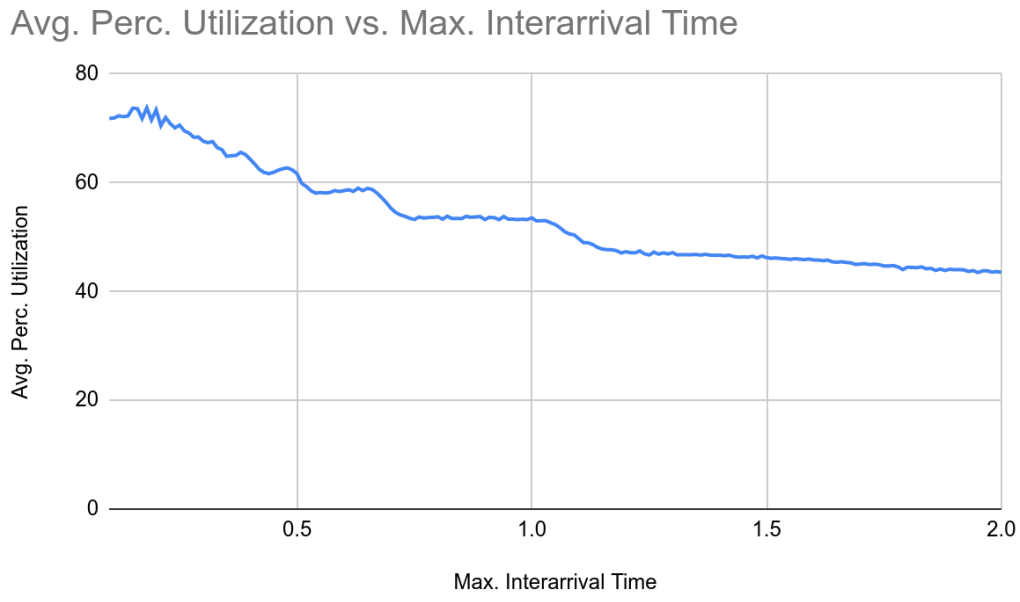

Avg. Perc. Utilization vs. Max. Interarrival Time
Avg. Perc. Utilization
100
75
50
25
0
0.5
1.0
1.5
2.0
Max. Interarrival Time

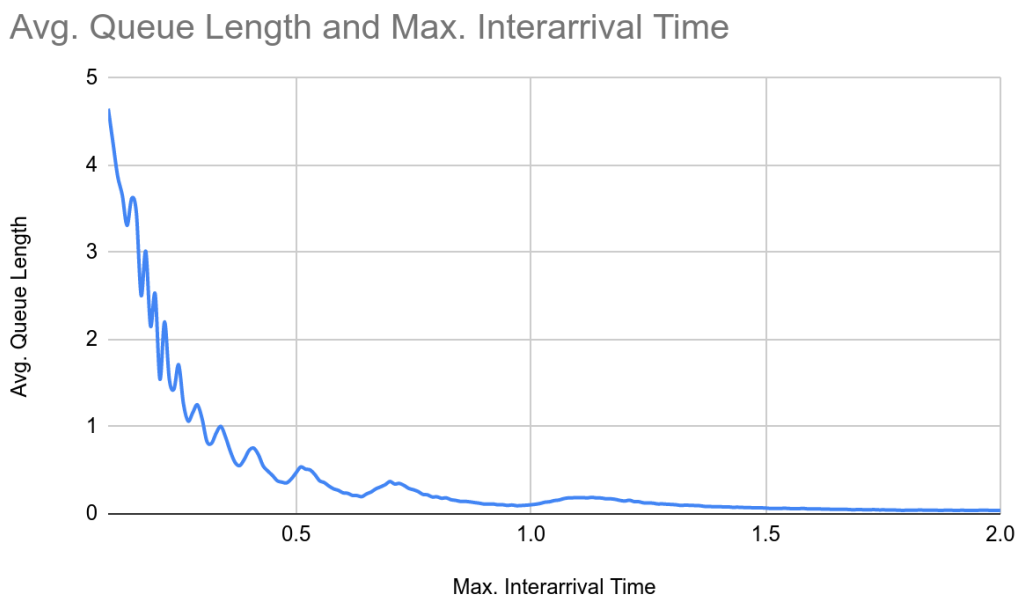


Channel 2 (Patients and Beds):




Avg. Delay vs. Max. Interarrival Time
0.25
0.20
0.15
0.10
0.05
0.00
Avg. Delay
0.5
1.0
1.5
2.0
Max. Interarrival Time


Avg. Perc. Utilization vs. Max. Interarrival Time
80
60
40
20
0
Avg. Perc. Utilization
0.5
1.0
1.5
2.0
Max. Interarrival Time



Trends observed:

- Both channels show similar trends for similar parameters.
- As expected, the number of ambulances, number of beds, average delay, average percentage utilization and average queue length have a decreasing trend overall.
- However, spikes and dips are observed. This is expected due to the maximum cap put on the number of deaths before simulation restarts.
- As maximum inter-arrival time changes, deaths $\in$ [0, maxDeaths) oscillates as maxDeaths-1, maxDeaths-2, ..., 1, 0, maxDeaths-1, maxDeaths-2, ...due to the simulation design aimed at predicting the absolute minimum possible value of ambulances/beds required.
- This discontinuity is responsible for the spikes and dips. As we reach the most loose state (deaths = 0), resources required are dropped leading to a sudden increase in the pressure on the system (deaths = maxDeaths-1)

Conclusion

The DES model can draw patterns between the population of the remote area and the required number of beds and ambulances. This would ensure a timely response to health issues avoiding amenable deaths. Future improvements to the model can help include more diverse aspects of health care like the availability of nurses, doctors, cleaners, equipment, machines, etc.

References

[1] Kruk, M.E, Gage, A.D., Joseph, N.T., Danaei, G., García-Saisó, S. & Salomon, J.A. (2018, September 05). Mortality due to low-quality health systems in the universal health coverage era: a systematic analysis of amenable deaths in 137 countries. The Lancet

https://www.thelancet.com/journals/lancet/article/PIIS0140-6736(18)31668-4/fulltext

[2] Law, A.M. (2014). Simulation Modeling and Analysis. McGraw-Hill Education. (1982)

[3] Banks, J., Carson, J.S., Nelson, B.L., Nicol, D.M., (2013). Discrete-Event System Simulation. Pearson. (1984)

Code

```
import java.util.ArrayList;
import java.util.Comparator;
import java.util.PriorityQueue;

public class Runner {
	private static final int numSimulation = 1000;
	private static final int maxNumDelay = 1000;

	private static ArrayList<ArrayList<Transfer>> allTransfers;
	private static double maxSeverity;
	private static int maxDeaths;

	private static class Transfer {
		double arrivalTime;
		double severity;

		Transfer(double arrivalTime, double severity) {
			this.arrivalTime = arrivalTime;
			this.severity = severity;
		}
	}

	private static class Channel1 {
		final double rateParameter;
		final double townRadius;
		final double maxInterArrivalTime;
		final double ambulanceSpeed;

		int numAmbulances;

		ArrayList<Transfer> transfers;
		int deaths;
		SystemState systemState;
		StatisticalCounters statisticalCounters;
		TimeProfile timeProfile;

		double avgNumAmbulances;
		double avgDelay;
		double avgQueueLength;
		double avgPercUtilization;
```

```
	Channel1(double rateParameter, double townRadius, double
maxInterArrivalTime, double ambulanceSpeed) {
		this.rateParameter = rateParameter;
		this.townRadius = townRadius;
		this.maxInterArrivalTime = maxInterArrivalTime;
		this.ambulanceSpeed = ambulanceSpeed;

		numAmbulances = 1;

		mainProgram();
	}

	class Patient {
		final double arrivalTime;
		double severity;
		final double distance;

		Patient() {
			arrivalTime = timeProfile.nextArrival;
			severity = exponentialLibraryRoutine() * maxSeverity;
			while (severity > maxSeverity) {
				severity = exponentialLibraryRoutine() *
maxSeverity;
			}
			distance = uniformLibraryRoutine() * townRadius;
		}
	}

	class Ambulance {
		Patient patient;
		double serviceTime;
		double delay;
		double departureTime;

		void admitPatient(Patient patient) {
			this.patient = patient;
			serviceTime = 2 * patient.distance / ambulanceSpeed;
			double timeServiceBegins = Math.max(patient.arrivalTime,
departureTime);
			delay = timeServiceBegins - patient.arrivalTime;
			departureTime = timeServiceBegins + serviceTime;
		}
	}
```

```
class SystemState {
	PriorityQueue<Patient> customers;
	ArrayList<Ambulance> servers;

	SystemState() {
		customers = new PriorityQueue<Patient>(new
Comparator<Patient>() {

			@Override
			public int compare(Patient patient1, Patient
patient2) {
				if (patient1.severity > patient2.severity)
{
					return -1;
				} else if (patient1.severity <
patient2.severity) {
					return 1;
				} else {
					return 0;
				}
			}

		});
		servers = new ArrayList<Ambulance>();
		for (int i = 0; i < numAmbulances; i++) {
			servers.add(new Ambulance());
		}
	}

	boolean updateSeverities() {
		double eventInterval = timeProfile.currClock -
timeProfile.prevClock;
		ArrayList<Patient> toRemove = new ArrayList<Patient>();
		for (Patient patient : customers) {
			patient.severity += eventInterval;
			if (patient.severity > maxSeverity) {
				deaths++;
				if (deaths > maxDeaths) {
					return true;
				}
				toRemove.add(patient);
			}
		}
		customers.removeAll(toRemove);
		return false;
```

```
                }

                Ambulance addPatient() {
                        Patient patient = new Patient();
                        customers.add(patient);
                        for (Ambulance ambulance : servers) {
                                if (ambulance.patient == null) {
                                        ambulance.admitPatient(customers.poll());
                                        return ambulance;
                                }
                        }
                        return null;
                }

                double serverStatus() {
                        int total = 0;
                        for (Ambulance ambulance : servers) {
                                if (ambulance.patient != null) {
                                        total++;
                                }
                        }
                        return (double) total / servers.size();
                }

                Ambulance removePatient() {
                        Ambulance ambulance = timeProfile.nextDepartures.peek();
                        transfers.add(new Transfer(ambulance.departureTime,
ambulance.patient.severity));
                        ambulance.patient = null;
                        if (customers.size() != 0) {
                                ambulance.admitPatient(customers.poll());
                                return ambulance;
                        }
                        return null;
                }
        }

        class StatisticalCounters {
                int numDelay;
                double totalDelay;
                double areaQt;
                double areaBt;

                void updateStatisticalCounters(Ambulance ambulance, int
prevQueueLength, double prevServerStatus) {
```

```
                        if (ambulance != null) {
                                numDelay++;
                                totalDelay += ambulance.delay;
                        }
                        double eventInterval = timeProfile.currClock -
timeProfile.prevClock;
                        areaQt += prevQueueLength * eventInterval;
                        areaBt += prevServerStatus * eventInterval;
                }
        }

        class TimeProfile {
                double prevClock;
                double currClock;
                double nextArrival;
                PriorityQueue<Ambulance> nextDepartures;

                TimeProfile() {
                        prevClock = -1;
                        nextArrival = uniformLibraryRoutine() *
maxInterArrivalTime;
                        nextDepartures = new PriorityQueue<Ambulance>(new
Comparator<Ambulance>() {

                                @Override
                                public int compare(Ambulance ambulance1, Ambulance
ambulance2) {
                                        if (ambulance1.departureTime >
ambulance2.departureTime) {
                                                return 1;
                                        } else if (ambulance1.departureTime <
ambulance2.departureTime) {
                                                return -1;
                                        } else {
                                                return 0;
                                        }
                                }

                        });
                }

                double updateClocks() {
                        prevClock = currClock;
                        if (nextDepartures.size() == 0) {
                                currClock = nextArrival;
```

```
                                return -1;
                        }
                        double nextDeparture =
nextDepartures.peek().departureTime;
                        currClock = Math.min(nextArrival, nextDeparture);
                        return nextArrival - nextDeparture;
                }

                void updateForArrival(Ambulance ambulance) {
                        nextArrival += uniformLibraryRoutine() *
maxInterArrivalTime;
                        if (ambulance != null) {
                                nextDepartures.add(ambulance);
                        }
                }

                void updateForDeparture(Ambulance ambulance) {
                        nextDepartures.poll();
                        if (ambulance != null) {
                                nextDepartures.add(ambulance);
                        }
                }
        }

        class ExitSimulationException extends Exception {
                static final long serialVersionUID = 1L;
        }

        class ResetSimulationException extends Exception {
                static final long serialVersionUID = 1L;
        }

        void initializationRoutine() {
                transfers = new ArrayList<Transfer>();
                deaths = 0;
                systemState = new SystemState();
                statisticalCounters = new StatisticalCounters();
                timeProfile = new TimeProfile();
        }

        double timingRoutine() {
                return timeProfile.updateClocks();
        }

        void arrivalEventRoutine() throws ResetSimulationException,
```

```
ExitSimulationException {
                    if (systemState.updateSeverities()) {
                          throw new ResetSimulationException();
                    }

                    int prevQueueLength = systemState.customers.size();
                    double prevServerStatus = systemState.serverStatus();
                    Ambulance ambulance = systemState.addPatient();

                    timeProfile.updateForArrival(ambulance);
                    statisticalCounters.updateStatisticalCounters(ambulance,
prevQueueLength, prevServerStatus);

                    if (statisticalCounters.numDelay == maxNumDelay) {
                          throw new ExitSimulationException();
                    }
             }

             void departureEventRoutine() throws ResetSimulationException,
ExitSimulationException {
                    if (systemState.updateSeverities()) {
                          throw new ResetSimulationException();
                    }

                    int prevQueueLength = systemState.customers.size();
                    double prevServerStatus = systemState.serverStatus();
                    Ambulance ambulance = systemState.removePatient();

                    timeProfile.updateForDeparture(ambulance);
                    statisticalCounters.updateStatisticalCounters(ambulance,
prevQueueLength, prevServerStatus);

                    if (statisticalCounters.numDelay == maxNumDelay) {
                          throw new ExitSimulationException();
                    }
             }

             double uniformLibraryRoutine() {
                    return Math.random();
             }

             double exponentialLibraryRoutine() {
                    return Math.log(1 - Math.random()) / (-rateParameter);
             }
```

```
            void calculateCounters() {
                  avgNumAmbulances += numAmbulances;
                  avgDelay += statisticalCounters.totalDelay /
statisticalCounters.numDelay;
                  avgQueueLength += statisticalCounters.areaQt /
timeProfile.currClock;
                  avgPercUtilization += (statisticalCounters.areaBt /
timeProfile.currClock) * 100;
                  allTransfers.add(transfers);
            }

            void reportGenerator() {
                  avgNumAmbulances /= numSimulation;
                  avgDelay /= numSimulation;
                  avgQueueLength /= numSimulation;
                  avgPercUtilization /= numSimulation;
            }

            void mainProgram() {
                  for (int i = 0; i < numSimulation; i++) {
                        numAmbulances = 1;
                        while (true) {
                              initializationRoutine();
                              try {
                                    while (true) {
                                          double event = timingRoutine();
                                          if (event < 0) {
                                                arrivalEventRoutine();
                                          } else {
                                                departureEventRoutine();
                                          }
                                    }
                              } catch (ResetSimulationException e) {
                                    numAmbulances++;
                                    continue;
                              } catch (ExitSimulationException e) {
                                    break;
                              }
                        }
                        calculateCounters();
                  }
                  reportGenerator();
            }

            void print() {
```

```
                    System.out.println("Channel 1:-");
                    System.out.printf("Number of Ambulances:\t%.4f\n",
avgNumAmbulances);
                    System.out.printf("Average Delay:\t\t%.4f\n", avgDelay);
                    System.out.printf("Average Queue Length:\t%.4f\n",
avgQueueLength);
                    System.out.printf("Percentage Utilization:\t%.4f\n",
avgPercUtilization);
                    System.out.println();
             }
      }

      private static class Channel2 {
             int numBeds;
             ArrayList<Transfer> transfers;

             int deaths;
             SystemState systemState;
             StatisticalCounters statisticalCounters;
             TimeProfile timeProfile;

             double avgNumBeds;
             double avgDelay;
             double avgQueueLength;
             double avgPercUtilization;

             Channel2() {
                    numBeds = 1;

                    mainProgram();
             }

             class Patient {
                    final double arrivalTime;
                    double severity;

                    Patient() {
                           arrivalTime = timeProfile.nextArrival;
                           severity =
transfers.get(timeProfile.arrivalPos).severity;
                    }
             }

             class Bed {
                    Patient patient;
```

```
                double serviceTime;
                double delay;
                double departureTime;

                void admitPatient(Patient patient) {
                        this.patient = patient;
                        serviceTime = patient.severity;
                        double timeServiceBegins = Math.max(patient.arrivalTime,
departureTime);
                        delay = timeServiceBegins - patient.arrivalTime;
                        departureTime = timeServiceBegins + serviceTime;
                }
        }

        class SystemState {
                PriorityQueue<Patient> customers;
                ArrayList<Bed> servers;

                SystemState() {
                        customers = new PriorityQueue<Patient>(new
Comparator<Patient>() {

                                @Override
                                public int compare(Patient patient1, Patient
patient2) {
                                        if (patient1.severity > patient2.severity)
{
                                                return -1;
                                        } else if (patient1.severity <
patient2.severity) {
                                                return 1;
                                        } else {
                                                return 0;
                                        }
                                }

                        });
                        servers = new ArrayList<Bed>();
                        for (int i = 0; i < numBeds; i++) {
                                servers.add(new Bed());
                        }
                }

                boolean updateSeverities() {
                        double eventInterval = timeProfile.currClock -
```

```
timeProfile.prevClock;
                        ArrayList<Patient> toRemove = new ArrayList<Patient>();
                        for (Patient patient : customers) {
                                patient.severity += eventInterval;
                                if (patient.severity > maxSeverity) {
                                        deaths++;
                                        if (deaths > maxDeaths) {
                                                return true;
                                        }
                                        toRemove.add(patient);
                                }
                        }
                        customers.removeAll(toRemove);
                        return false;
                }

                Bed addPatient() {
                        Patient patient = new Patient();
                        customers.add(patient);
                        for (Bed bed : servers) {
                                if (bed.patient == null) {
                                        bed.admitPatient(customers.poll());
                                        return bed;
                                }
                        }
                        return null;
                }

                double serverStatus() {
                        int total = 0;
                        for (Bed bed : servers) {
                                if (bed.patient != null) {
                                        total++;
                                }
                        }
                        return (double) total / servers.size();
                }

                Bed removePatient() {
                        Bed bed = timeProfile.nextDepartures.peek();
                        bed.patient = null;
                        if (customers.size() != 0) {
                                bed.admitPatient(customers.poll());
                                return bed;
                        }
```

```
                        return null;
                }
        }

        class StatisticalCounters {
                int numDelay;
                double totalDelay;
                double areaQt;
                double areaBt;

                void updateStatisticalCounters(Bed bed, int prevQueueLength,
double prevServerStatus) {
                        if (bed != null) {
                                numDelay++;
                                totalDelay += bed.delay;
                        }
                        double eventInterval = timeProfile.currClock -
timeProfile.prevClock;
                        areaQt += prevQueueLength * eventInterval;
                        areaBt += prevServerStatus * eventInterval;
                }
        }

        class TimeProfile {
                double prevClock;
                double currClock;
                int arrivalPos;
                double nextArrival;
                PriorityQueue<Bed> nextDepartures;

                TimeProfile() {
                        prevClock = -1;
                        nextArrival = transfers.get(arrivalPos).arrivalTime;
                        nextDepartures = new PriorityQueue<Bed>(new
Comparator<Bed>() {

                                @Override
                                public int compare(Bed bed1, Bed bed2) {
                                        if (bed1.departureTime >
bed2.departureTime) {
                                                return 1;
                                        } else if (bed1.departureTime <
bed2.departureTime) {
                                                return -1;
                                        } else {
```

```
                                        return 0;
                                    }
                                }

                        });
                }

                double updateClocks() {
                        prevClock = currClock;
                        if (nextDepartures.size() == 0) {
                                currClock = nextArrival;
                                return -1;
                        }
                        double nextDeparture =
nextDepartures.peek().departureTime;
                        currClock = Math.min(nextArrival, nextDeparture);
                        return nextArrival - nextDeparture;
                }

                void updateForArrival(Bed bed) {
                        arrivalPos++;
                        nextArrival = transfers.get(arrivalPos).arrivalTime;
                        if (bed != null) {
                                nextDepartures.add(bed);
                        }
                }

                void updateForDeparture(Bed bed) {
                        nextDepartures.poll();
                        if (bed != null) {
                                nextDepartures.add(bed);
                        }
                }
        }

        class ExitSimulationException extends Exception {
                static final long serialVersionUID = 1L;
        }

        class ResetSimulationException extends Exception {
                static final long serialVersionUID = 1L;
        }

        void initializationRoutine() {
                deaths = 0;
```

```
                systemState = new SystemState();
                statisticalCounters = new StatisticalCounters();
                timeProfile = new TimeProfile();
        }

        double timingRoutine() {
                return timeProfile.updateClocks();
        }

        void arrivalEventRoutine() throws ResetSimulationException,
ExitSimulationException {
                if (systemState.updateSeverities()) {
                        throw new ResetSimulationException();
                }

                int prevQueueLength = systemState.customers.size();
                double prevServerStatus = systemState.serverStatus();
                Bed bed = systemState.addPatient();

                if (timeProfile.arrivalPos == transfers.size() - 1) {
                        throw new ExitSimulationException();
                }

                timeProfile.updateForArrival(bed);
                statisticalCounters.updateStatisticalCounters(bed,
prevQueueLength, prevServerStatus);
        }

        void departureEventRoutine() throws ResetSimulationException,
ExitSimulationException {
                if (systemState.updateSeverities()) {
                        throw new ResetSimulationException();
                }

                int prevQueueLength = systemState.customers.size();
                double prevServerStatus = systemState.serverStatus();
                Bed bed = systemState.removePatient();

                if (timeProfile.arrivalPos == transfers.size() - 1) {
                        throw new ExitSimulationException();
                }

                timeProfile.updateForDeparture(bed);
                statisticalCounters.updateStatisticalCounters(bed,
prevQueueLength, prevServerStatus);
```

```
            }

            void calculateCounters() {
                  avgNumBeds += numBeds;
                  avgDelay += statisticalCounters.totalDelay /
statisticalCounters.numDelay;
                  avgQueueLength += statisticalCounters.areaQt /
timeProfile.currClock;
                  avgPercUtilization += (statisticalCounters.areaBt /
timeProfile.currClock) * 100;
            }

            void reportGenerator() {
                  avgNumBeds /= numSimulation;
                  avgDelay /= numSimulation;
                  avgQueueLength /= numSimulation;
                  avgPercUtilization /= numSimulation;
            }

            void mainProgram() {
                  for (int i = 0; i < numSimulation; i++) {
                        numBeds = 1;
                        transfers = allTransfers.get(i);
                        while (true) {
                              initializationRoutine();
                              try {
                                    while (true) {
                                          double event = timingRoutine();
                                          if (event < 0) {
                                                arrivalEventRoutine();
                                          } else {
                                                departureEventRoutine();
                                          }
                                    }
                              } catch (ResetSimulationException e) {
                                    numBeds++;
                                    continue;
                              } catch (ExitSimulationException e) {
                                    break;
                              }
                        }
                        calculateCounters();
                  }
                  reportGenerator();
            }
```

```
		void print() {
			System.out.println("Channel 2:-");
			System.out.printf("Number of Beds:\t\t%.4f\n", avgNumBeds);
			System.out.printf("Average Delay:\t\t%.4f\n", avgDelay);
			System.out.printf("Average Queue Length:\t%.4f\n",
avgQueueLength);
			System.out.printf("Percentage Utilization:\t%.4f\n",
avgPercUtilization);
			System.out.println();
		}
	}

	private Runner() {
	}

	public static void run() {
		allTransfers = new ArrayList<ArrayList<Transfer>>();
		maxSeverity = 10;
		maxDeaths = 5;

		double rateParameter = 4;
		double townRadius = 30;
		double maxInterArrivalTime = 0.5;
		double ambulanceSpeed = 30;

		Channel1 c1 = new Channel1(rateParameter, townRadius,
maxInterArrivalTime, ambulanceSpeed);
		c1.print();

		Channel2 c2 = new Channel2();
		c2.print();
	}

	public static void example() {
		maxSeverity = 10;
		maxDeaths = 5;

		double rateParameter = 4;
		double townRadius = 30;
		double ambulanceSpeed = 30;

		for (double i = 0.1; i <= 2.01; i += 0.01) {
			allTransfers = new ArrayList<ArrayList<Transfer>>();
```

```
                    double maxInterArrivalTime = i;

                    Channel1 c1 = new Channel1(rateParameter, townRadius,
maxInterArrivalTime, ambulanceSpeed);
                    Channel2 c2 = new Channel2();
                    System.out.printf("%.4f, %.4f, %.4f, %.4f, %.4f, %.4f, %.4f,
%.4f, %.4f\n", maxInterArrivalTime,
                                    c1.avgNumAmbulances, c1.avgDelay,
c1.avgPercUtilization, c1.avgQueueLength, c2.avgNumBeds,
                                    c2.avgDelay, c2.avgPercUtilization,
c2.avgQueueLength);
             }
      }

      public static void main(String[] args) {
             Runner.run();
//           Runner.example();
      }
}
```