Use of AKKA Streams to Inform Forward Simulation in Robot Control

Peter Böhm[1] and Surya P. N. Singh[2]
[1]ACME University, Australia
[2]Robotics Design Lab, The University of Queensland, Australia
peter@nextlogic.biz, spns@uq.edu.au

Abstract

We investigate the use of AKKA Streams to parallelize robot simulation. This allows for parallel processing of data streams generated by multiple sensors on multiple robots in situations where it is not possible to pause the simulation during calculations and network communication. At the same time, it encapsulates entire simulation into a self-contained set of processes. This, in turn, allows for multiple cases to be running in parallel, which may be beneficial for problems requiring forward simulation. This is seen in stochastic multirobot optimization and control problems solved using reinforced learning (RL). Use of Akka streams helps to mitigate the effects of network and calculations latency and provides an expressive DSL (Domain Specific Language) to define the stream transformation graphs.

1 Introduction

Typically, the reinforcement learning works with a simulator. The simulator provides observations, and then a learning algorithm has a policy which chooses an action. The simulator then processes the action and returns a reward. This simple time loop repeats until the end of the episode at which point there are more calculations (e.g. the policy is updated) and everything starts again. Because the simulator is on hold during the calculations and data transfer, there is no latency to be considered, the calculations use accurate state observations of the environment updated after every turn. When dealing with multiple agents, they are all updated at the same moment before the simulation resumes, and as such there is no delay and no inter-dependencies on the order of the agents' actions. This is shown in Figure 1.

There are many high fidelity simulators that provide accurate modeling of physics, such as AirSim [Shah *et al.*, 2017], Gazebo and OpenAI Gym. The training of

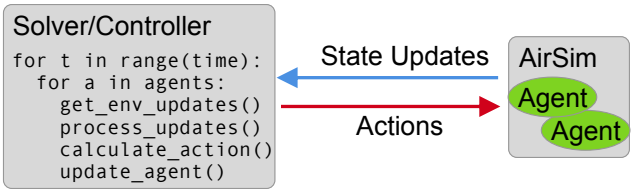

Figure 1: Example of a simple solver interacting with the simulator.

the RL models is, in most cases, dependent on these assumptions of no latency and data accuracy. Situations in which it is not possible to stop the simulation during calculations and those that deal with remote agents (e.g. using a cloud GPU) require a different approach. Because both calculations and network communication involve certain latency, dealing with multiple sensors attached to multiple agents can result in material delays between receiving and processing the data and delivering the actions back to the agents. This is demonstrated in Figure 4 and Figure 5. The images represent trajectories generated by pursue flight. The blue color represents the evader and the red color represents the evader. Trajectories in all of the images are using the same algorithm. The first image in Figure 4 represents the ground truth. After every step, the simulator was paused for location updates, calculations and communication of actions back to the agents. The rest of the images show results without pausing the simulator which caused various degrees of latency, depending on the level of parallelization. Figure 5 shows performance of the same algorithm using network connection with higher and much more variable latency.

While it is possible to reduce the communication delay by running everything on a single machine and thus eliminating the expensive network communication and reduce the calculation delay by simply using a more powerful machine, this approach may not scale sufficiently and it may not even be possible if external hardware is in the loop. E.g. the AirSim alone requires a modern GPU,

significant amount of RAM and at least 8 core CPU. Running more computationally expensive tasks such as image recognition to process the camera feed or deep learning of policy gradient along side the rendering engine is not possible without compromising the stability of the entire system.

We propose a different solution using parallelization. Instead of handling the tasks sequentially within a single time loop, they can be processed in parallel. This means that there is a separate sub-process [1] handling updates from each sensor, separate sub-processes handling different parts of calculations and separate sub-processes handling communication of actions back to the agents. Indeed, even the processes within a single pipe-line can be parallelized (e.g. image recognition on the camera feed can be processed in parallel as soon as the data are received so the next image can start processing before the previous has completed).

Handling and processing multi-channel communication with multiple robots is a necessary stepping stone to bridging the gap between simulation and reality. *citations needed - there's plenty papers on bridging the gap*

2 Software Framework

2.1 Simulator

This text describes integration with AirSim. AirSim is a simulator for drones, cars and more, built on Unreal Engine. It is open-source, cross platform, and supports hardware-in-loop with popular flight controllers such as PX4 for physically and visually realistic simulations. It is developed as an Unreal plugin that can simply be dropped into any Unreal environment. [Shah *et al.*, 2017]

2.2 Akka and Actor Model

Actor abstraction is a parallel programming model based on actors and messages. Immutable messages are passed between actors asynchronously and delivered into the recipient's mailbox. Each message is then completed in a single threaded fashion. This is also true for the internal state of the actor - it can only be modified and accessed by the actor [Lawlor and Walsh, 2016]. Actor systems easily express a wide range of computational paradigms, and provide a natural extension of programming into concurrent (parallel) systems. This naturalness and ease of expression means that programs to solve complex problems do not add even more complexity of their own. This direct relationship of the code to the problem domain also makes it easier to optimize algorithms based on knowledge of that domain [Agha *et al.*, 1997].

[1] No specific implementation is implied here. The process is not tied to the specific processor core or thread.

Communication with AirSim

AirSim uses MessagePack [Furuhashi, 2014] for communication. MessagePack is an RPC (Remote Procedure Call) implementation that provides fast and efficient communication between components written in different languages. At the time of writing, the last update of the JVM implementation was done 7 years ago[2] and it also depends on a no longer supported version of MessagePack serializer[3]. The RPC implementation doesn't provide any error handling which makes it impossible to properly handle errors returned by AirSim. This implementation cannot be used for camera feed because the images returned by `simGetImage` call exceed the allowed value size. Lastly, mapping between its Value types and native Scala types is either missing or is problematic and slow.

As such the a custom RPC implementation using a routing pool of Akka TCP clients was used. The solver uses ask messages to to send commands and receive responses in a non-blocking manner. A Scala msgpack4s[4] library was used for serialization/de-serialization.

2.3 Reactive Streams

High fidelity simulators, just like real-world implementations, generate large amount of data that needs to be processed in near real-time. Each agent can generate multiple streams (e.g. location updates, camera feed, LIDAR feed, etc.) and streams from multiple agents may need to combined (e.g. in synchronized movement of multiple vehicles). Each stream may need multiple transformations (e.g. deserialization, conversion, calculations, etc.) and those transformations may need to be parallelized to increase throughput. Reactive Streams is an initiative to provide a standard for asynchronous stream processing with non-blocking back pressure. This encompasses efforts aimed at runtime environments (JVM and JavaScript) as well as network protocols. [rea, 2019]

Back pressure

In a system without back pressure, if the subscriber is slower than the publisher, then eventually the stream will stop - either because one of the parties runs out of memory and one of its buffers overflows or because the implementation can detect this situation but cannot stop sending or accepting data until the situation is resolved (if it can be resolved at all). Although the latter scenario is somewhat more positive than the former, blocking back pressure (the capability of a system to detect when it must slow down and to do so by block-

[2] https://github.com/msgpack-rpc/msgpack-rpc-java

[3] Version 0.6.6 of MessagePack for Java (https://github.com/msgpack/msgpack-java)

[4] https://github.com/velvia/msgpack4s

ing) brings with it all of the disadvantages of a blocking system, which occupies resources such as thread and memory usage. [Bernhardt, 2016]

2.4 Domain Specific Language (DSL)

Besides technical reasons, there is also additional motivation for the reactive stream. They provide a DSL (Domain Specific Language) [Fowler, 2010] to define the stream transformation graphs in a expressive way. In a few lines of code it's very easy to express how the data sources are transformed and how data stream flows between the components and where it's terminated.

This is illustrated in Figure 2. Location streams for both actors are merged together using a merge component and then fed into the relative distance calculator. The result is then broadcasted into 2 parallel streams, one for each agent. Based on the relative position, the solver calculates actions in the form of steering thetas. Each steering theta is then broadcasted into 3 parallel streams - the first one to update the AirSim, second to feed the updated theta back to the solver and the last one to persist the steering decision data.

2.5 Message Broker

For situation where the solver cannot be implemented in the same program (e.g. because it is implemented in a different programming language) or on the same machine (e.g. because it requires extra resources such as GPU), it is necessary to implement remote messaging.

One way is to use RPC (Remote Procedure Call). There are several options available: AirSim uses MessagePack RPC (Remote Procedure Call), Google uses gRPC for its services, and there are several others. Another option is to use a message broker. The difference is that while RPC calls the remote service directly, with message broker, the services publish and consume messages via the message broker.

Kafka was developed to collect and distribute massive messages for distributed systems. It is a high-throughput distributed messaging system. Kafka can keep stable performance even if it processes millions of messages per second. [Wang *et al.*, 2015]

Figure **??** shows how such communication works. The raw (or pre-processed) data is published to Kafka (e.g. one topic for location updates, one topic for LIDAR, etc.) and consumed by services that process these data. The results are then published to a different topic. This simplifies dependencies between services, reduces data loss when a service crashes, provides a simplicity of one "API" for communication.

3 Experiment

The same algorithm was tested in 4 different setups - sequential model with simulator paused for calculations and communication (this represents the ground truth), sequential model with simulator running throughout, partially parallelized model using futures to encapsulate communication/calculations for each agent at every time step, and fully parallelized model using Akka Streams. Each model was run under 2 different network setups[5]. The first set of experiments was using a direct internet connection which provides stable average latency of 25ms. The second set was run over VPN with average latency of 70ms, fluctuating between 25ms and 70ms (Figure 3 shows the latency details).

A fixed algorithm was chosen as a policy to avoid side-effects that could be introduced by stochastic nature of RL training *this could back-fire by leading back to the magic RL that can compensate for everything.* The algorithm was originally created for simulation of the differential games [Redulla and Singh, 2018]. The pursuer is faster, moving with velocity of 10m/s, but has a larger turning radius of 8m. The evader is slower at velocity of 5m/s, but is more agile and can make turns of any angle. The algorithm only requires position of both players and their headings[6].

Four observations were measured: recency of location data of the player and opponent and time since the last move by the player and opponent.

3.1 Sequential model with simulator paused between turns

This represents the ground truth as the paused simulator serves as a synchronization layer that shields the algorithm from both stale data and delays in actions. The simulator starts paused. The measurements are taken, and the actions calculated and sent to the agents. After that, the simulator is run for 100ms. Then the process repeats. Because the communication with the simulator happens in the paused state it can be done in a blocking manner and the calculations start only once all the requests have finished.

3.2 Sequential model with simulator running

This is the same as the previous model, but without the paused simulator acting as the synchronization layer. Because all commands run in sequence, the next request can only be sent once the previous one completes and the calculations can only start once all the requests finished. This creates a varying level of staleness among the received data that enter the calculations. It also increases

[5]The simulator is running on a GPU on GCP (Google Cloud Platform) located in Sydney (zone australia-southeast1) while the solver was located in Brisbane (about 1000km away).

[6]The location data for each player is shared in every implementation to reduce the number of requests because high number of concurrent requests tends to overload AirSim.

```
eLocationsE ~> merge
pLocationsE ~> merge

merge ~> relativeDistanceFlow(eRelPositionActor) ~>
  broadcastRelDistance ~> calculateEvadeTheta ~> eBroadcast ~> evadeAirSim(airSimPoolMaster)
                                                 eBroadcast ~> updateTheta(Constants.e, eRelPositionActor)
                                                 eBroadcast  ~> eSaveSteeringDecision
  broadcastRelDistance ~> calculatePursueTheta ~> pBroadcast ~> pursueAirSim(airSimPoolMaster)
                                                  pBroadcast ~> updateTheta(Constants.p, eRelPositionActor)
                                                  pBroadcast ~> pSaveSteeringDecision
```

Figure 2: The graph controlling the simulation expressed using the Akka Streams DSL. Symbol ~> stands for "feeds into".

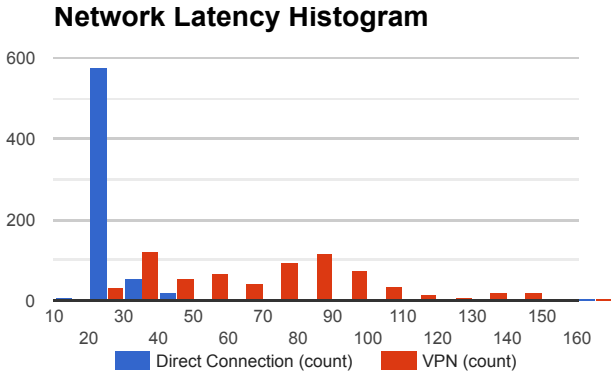

Figure 3: Histogram of the latency using direct connection and VPN.

the time lag between actions being sent to the agents. With the pursuer moving at 10m/s, even relatively small delay of 500ms translates to a location difference of 5m.

3.3 Partially parallelized model

This model represents a simple (if short-sighted) solution to the concurrency issue. Since each agent is treated as a separate entity, at each timestamp, they are wrapped in their individual future. This means that while all communication and calculations for a single agent runs sequentially, the code blocks for both agents are run in parallel.

3.4 Fully parallelized model using Akka Streams

In this model, not only the agents move in parallel, but each operation within each agent is parallelized as well. There are no inter-process dependencies: the location is updated independently, the calculations use whatever latest data available and once the new action is calculated, it is sent to the agent. The whole process moves in the interleaved manner. *this whole paragraph needs to be changed*

4 Results

Figures 4 and 5 show the summary of the results. The first column illustrates the trajectories generated by the agents. Because this is a pursue/evade situation, the distance between the agents is an important measure. This is plotted in the charts in the second column. Charts in the third column show recency of data used for calculations and time-lag since the last move.

4.1 Direct Connection

The first image shows the ground truth with the drones orbiting after about 11s. The distance between the agents converges fast to about 8.55m. The sequential model (model 2) performs the worst. The delays caused by the sequential execution of the blocking calls to update location details cause much longer turning times. This is especially visible in the pursuer because it's moving faster and any delay in control will have it move further off course. The result is the distance alternating between large and small values. The partially parallelized model performs reasonably well, the trajectory follows the ground truth closely, albeit with a larger diameter. The distance between the agents converges to about 10m. Because both location updates run in parallel, they finish on average in 25ms and the action sent to back to the simulator is delayed by the same 25ms as compared to the ground truth (see the Time Since My Last Move chart). This delay causes a minor drift in agents which is what causes the slightly larger radius. Model 4 follows the ground truth the most closely. The distance converges to 9.1 which results in a slightly larger diameter but smaller than model 3. The delays between moves are very close to 100ms just like in the ground truth model.

4.2 VPN Connection

Under the more demanding conditions of the VPN environment, only the fully parallelized model had acceptable performance.

5 Conclusions

Acknowledgments

The preparation of these instructions and the LaTEX and BibTEX files that implement them was supported by Schlumberger Palo Alto Research, AT&T Bell Laboratories, and Morgan Kaufmann Publishers.

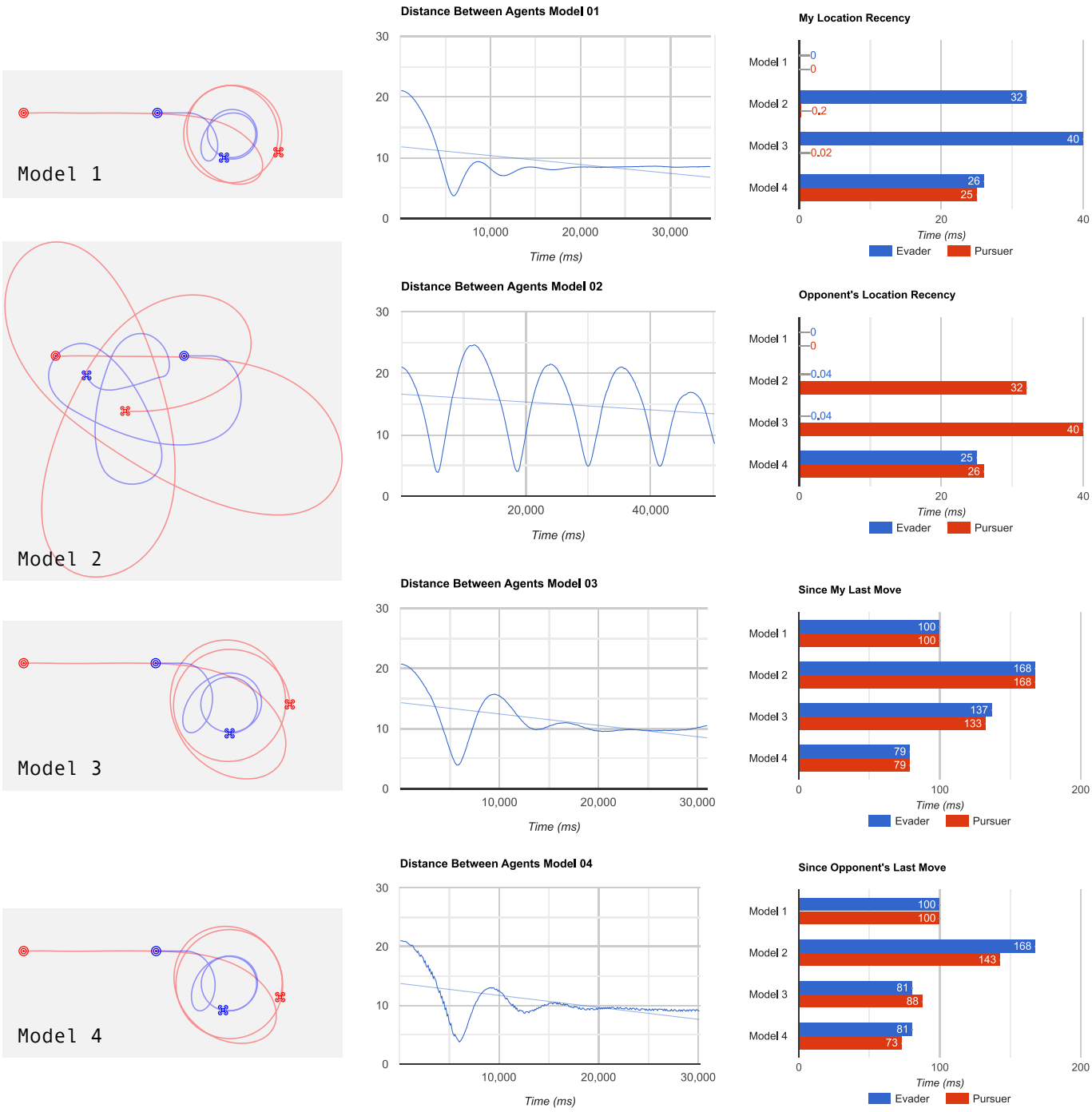

Figure 4: Results of running the models over a reliable network with latency averaging 25ms. The first column shows the trajectory followed by the agents. The second column shows the distance between the agents. The third column shows the recency of the data used for calculations and delay since the previous step.

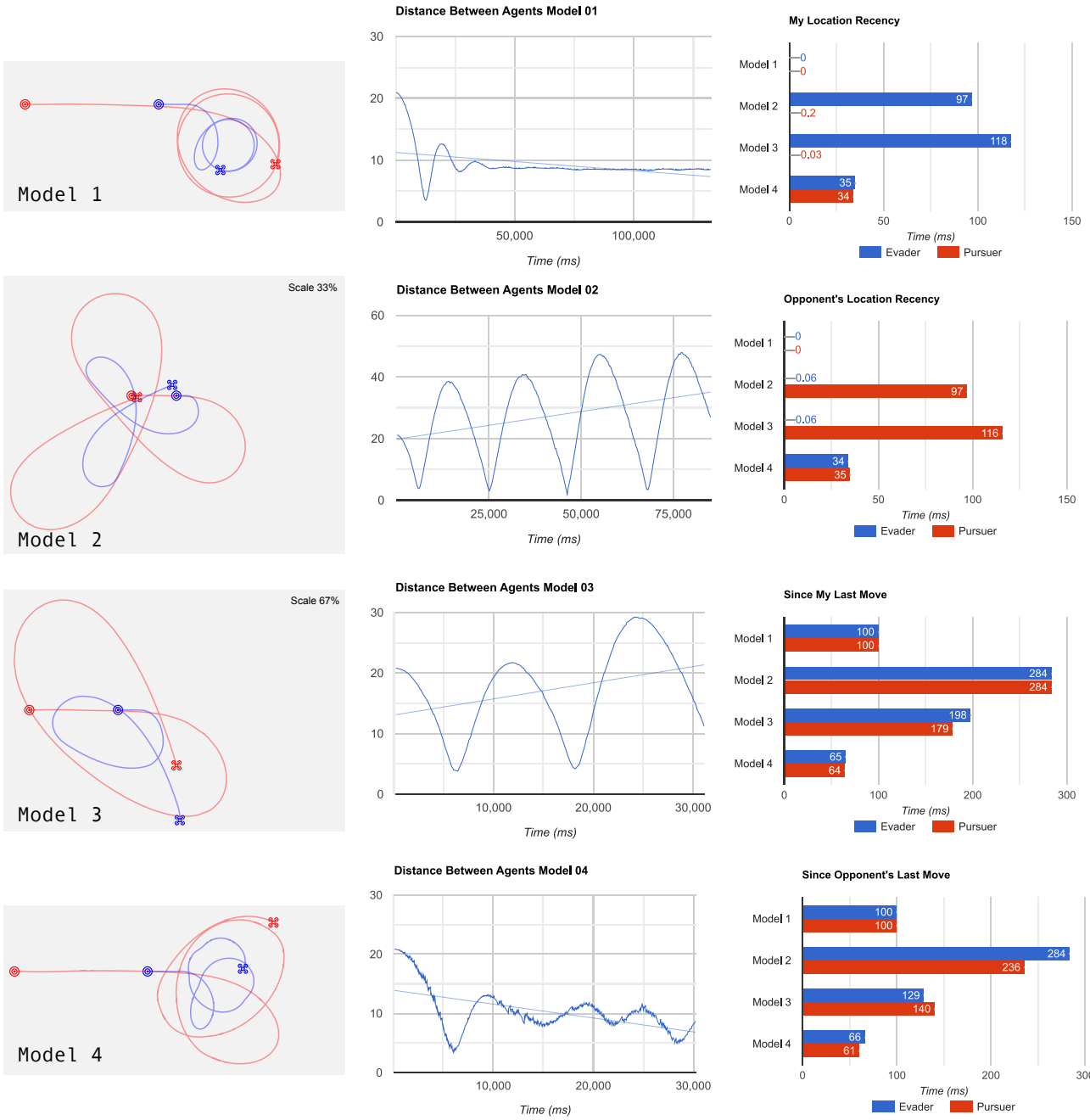

Figure 5: Results of running the models over a VPN with fluctuating latency (average of 70ms). The first column shows the trajectory followed by the agents. The second column shows the distance between the agents. The third column shows the recency of the data used for calculations and delay since the previous step.

References

[Agha *et al.*, 1997] Gul A Agha, Ian A Mason, Scott F Smith, and Carolyn L Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(1):1–72, 1997.

[Bernhardt, 2016] Manuel Bernhardt. *Reactive Web Applications: Covers Play, Akka, and Reactive Streams*. Manning Publications Co., 2016.

[Fowler, 2010] Martin Fowler. *Domain-specific languages*. Pearson Education, 2010.

[Furuhashi, 2014] S Furuhashi. Messagepack.(2017). *URL http://msgpack. org/. Accessed*, pages 02–21, 2014.

[Lawlor and Walsh, 2016] Brendan Lawlor and Paul Walsh. The weekend warrior: how to build a genomic supercomputer in your spare time using streams and actors in scala. In *2016 Intl IEEE Conferences on Ubiquitous Intelligence & Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People, and Smart World Congress (UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld)*, pages 575–580. IEEE, 2016.

[rea, 2019] 2019.

[Redulla and Singh, 2018] Anne Redulla and Surya PN Singh. Simulating differential games with improved fidelity to better inform cooperative & adversarial two vehicle uav flight. In *2018 IEEE International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*, pages 130–136. IEEE, 2018.

[Shah *et al.*, 2017] Shital Shah, Debadeepta Dey, Chris Lovett, and Ashish Kapoor. Airsim: High-fidelity visual and physical simulation for autonomous vehicles. In *Field and Service Robotics*, 2017.

[Wang *et al.*, 2015] Zhenghe Wang, Wei Dai, Feng Wang, Hui Deng, Shoulin Wei, Xiaoli Zhang, and Bo Liang. Kafka and its using in high-throughput and reliable message distribution. In *2015 8th International Conference on Intelligent Networks and Intelligent Systems (ICINIS)*, pages 117–120. IEEE, 2015.