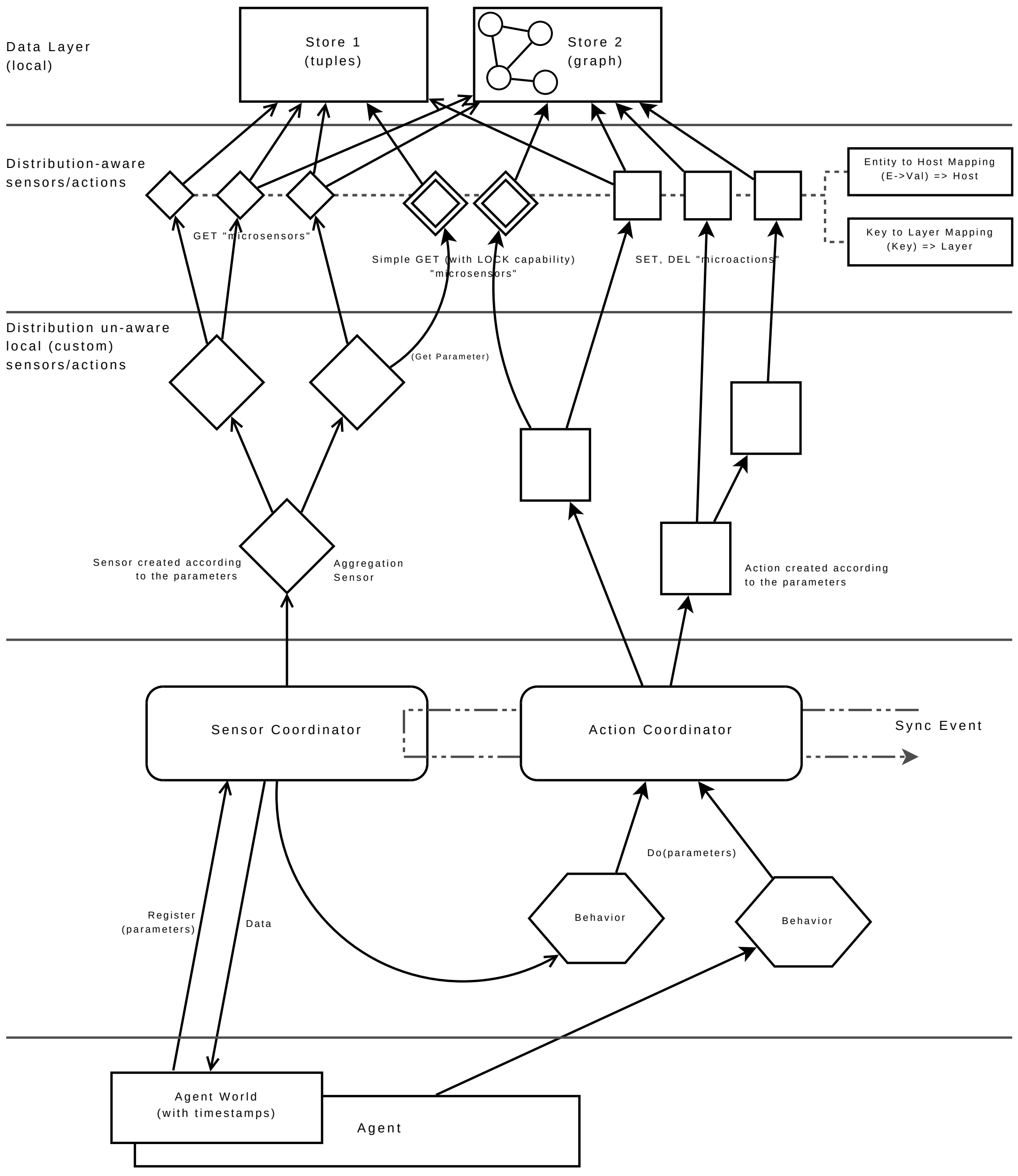
1 Distributed World Model for ATG Simulations

This text provides a new concept of distributed world model that should be suitable for most of the simulations developed at ATG. Using the common model should make the development more efficient and allow for straightforward integration of the simulators. We do not discuss the reasons for the presented architecture, the goal of this document is to summarize the decisions and serve as a base that we agree on.

The "Big Picture" of the model is shown in the following picture.

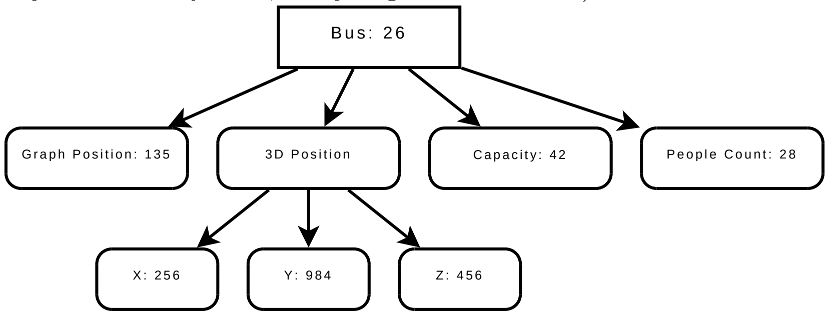


Data Layer (local)
Store 1 (tuples)
Store 2 (graph)
Distribution-aware sensors/actions
Entity to Host Mapping (E->Val) => Host
Key to Layer Mapping (Key) => Layer
GET "microsensors"
Simple GET (with LOCK capability) "microsensors"
SET, DEL "microactions"
Distribution un-aware local (custom) sensors/actions
(Get Parameter)
Sensor created according to the parameters
Aggregation Sensor
Action created according to the parameters
Sensor Coordinator
Action Coordinator
Sync Event
Do(parameters)
Behavior
Behavior
Register (parameters)
Data
Agent World (with timestamps)
Agent

1.1 Data Layer

Data Layer provides a general storage for the world items and entities. This layer is unaware of the distribution, it just saves data on a single host where it is running; the layer can contain more *stores* - those are the separate modules optimized for various data types. The idea about the data organization is following:

- The data is always about some entity in the world. Therefore, the main concept is to store these entities and than some details about them.
- The data about an entity could reside in several *store*s.
- The data about an entity is organized in a tree — all the entities form a forest. The children of the entity root node represent various entity properties (color, graph node where the entity is located, passengers count, but also some ”compound” properties like 3D position consisting of three coordinates - these might be leaves at 3rd graph level). The following figure shows an example of such properties. (We use the term *entity key* while talking about the root of a tree and *entity data* means the whole tree that represents this entity. By the *key* we mean any node, a *key* might have a *value.*)



- Each *key* can hold a primitive type value. Each node has children with unique ¡*key-type*, *value*¿ pair. A *key-type* needs to preserve a single *value-type* within all the stores (among all the computers).
- Except the root of the trees (entity node), a single node-type can reside in a single store module only – i.e. every entity property can be stored in a single store. In other words, the the relation (node-type: store module) is injective.

Every store provides the GET, SET, and DEL primitives for data manipulation:

GET has the following semantics:

$GET(E[\rightarrow Val]; \langle K_1[\rightarrow V_1], ..., K_{n_1}[\rightarrow V_{n_1}]\rangle; \langle ...\rangle)$

The store matches this tree-like query to the data and returns a list of the subtrees filled in with missing *values* in the query. Based on the constraints above (one *entity* on one computer, a single *key type* in a single store) the match can be done very quickly — the first (or any) key in the angle brackets tells us what store we need to use for this sequence and the store does not need to find the data - it just ”walks down” the tree for a specific *entity* and checks the presence/absence of a *key* (possibly with a specific value).

GET value parts (V_x) could have one of the following format:

– single value

– a range (two values), a returned value should fit in between including both delimiters

– a pair $\langle value, distance\rangle$, a returned value's distance from the value passed in need to be less or equal to the distance specified

SET is very similar to **GET**, except that all the values are mandatory and they can contain just a single value, not a range or value-distance pair:

$SET(E \rightarrow Val; \langle K_1 \rightarrow V_1, ..., K_{n_1} \rightarrow V_{n_1} \rangle; \langle ... \rangle)$

SET works also as **INSERT** that the store doesn't support — setting a key that is not present means an addition it to the store.

DEL looks also similar – except that all the values need to be specified except the last one in the key sequence; omitting the last value deletes all the keys of that type. The special case is the following:
$DEL(E)$
which deletes all the entities of type E, while
$DEL(E, value)$
deletes just a single entity of type E determined by the value. That's why we need DEL primitive and it is not sufficient to just SET some *null* values – we need to differentiate the two above cases.

Each store needs to support also locking mechanism used by the upper layers — every *key* needs a capability to hold two values (current one and a possible new one) and a lock if the key is currently locked. Moreover, also deletion (locked, with possible rollback) needs to be supported, so a flag whether the node (with all its children) is about to be deleted is required. Both READ and WRITE locks are supported with a usual semantics. Two additional functions are necessary for unlocking:

COMMIT $COMMIT(LockNo.)$ applies all the changes that are locked by this number and unlocks the keys.

ROLLBACK $ROLLBACK(LockNo.)$ discards all the changes and unlocks the keys.

1.2 Distribution-aware Layer

The second layer forms an abstraction that hides the distribution of stores among several computers. The interface is exactly the same as provided by the individual *store*s — however, it checks on which node the entity is located and possibly delegates the query/modification to a remote host. The semantics of the interface allows for specifying quite powerful (but quickly executable) queries and therefore most of the data might be filtered on a remote side and only the results that we probably need on our side are transmitted over the network. This layer contains three types of objects:

- Full Getters

  provide a capability to perform a full distributed query to a GET request described above. The result format is a list of subtrees with all the keys containing current values; it is than passed further to the sensor that made the request for further processing/filtration.

- Simple Getters with locking capability

  A Getter that accepts only a query where entity-key value is present and all the sequences of keys contain values except the last key. This last key has to be unique in the path – this ensures that just a single subtree will be returned:

  $GET(E \rightarrow Val; \langle K_1 \rightarrow V_1, ..., K_{n_1-1} \rightarrow V_{n_1-1}, K_{n_1} \rangle; \langle ... \rangle; [LockNo.; WRITE])$

  Moreover, this GET accepts two optional parameters:

  - *Lock No.* that supports actions – this locks the keys that contain the values returned for reading with a lock identified by the number.

  - READ/WRITE lock specification – some parameters retrieved by get are not changed during the action execution; they should be locked for reading only, allowing other simultaneous actions also to read the value.

  Usage of this locking will be described in the next layer.

- Setters (that can also delete)

  Setters accept the SET and DEL queries described above and execute them in a distributed manner. In addition, they propagate a Lock No. and the changes are not applied immediately — they are persisted by using COMMIT or discarded by ROLLBACK.

Since the setters/getters work in a distributed environment, they need to have some knowledge about the distribution. Especially:

- Where the entity data is located.
- Which store to look in for a specific keys.

This information is provided by two additional objects:

- **Entity to Host Mapping** – each host has a copy of this ”database”, it is synchronized among all the hosts. We do not allow entity data migration in the first version, therefore the modifications involve just insert and delete, there is no update - this simplifies the distribution management.
- **Key to Store Mapping** is static or almost static during the simulation, only rare inserts are expected (in case a new key is created). Therefore a distribution of this information is relatively simple.

1.3 Distribution Un-Aware Local Custom Sensors/Actions

This layer is built atop of the getters/setters. Since they handle the distribution, the sensors/actions do not need to care about it. However, it might not always be the most efficient way – especially for the sensors under the following conditions:

- The sensor computation is very expensive and exactly the same sensor is present on two or more computers. In such a case it might be better to perform the computation on a single box and distribute just the results.
- The *getter* interface is not rich enough for the sensor and the sensor needs to get much more data (possibly distributed among several machines) than is actually used and provided to the sensor listeners. This kind of sensor might force the lower layers to transport huge amount of data in between the computers even if just transfer to a single host where the computation would be performed followed by results distribution would be much more efficient.

We do not support the *distribution-aware* sensors in this version, it might be added in the following releases.

1.3.1 Sensors

A sensor object is an instance of a sensor created with some parameters. These parameters cannot change, a separate object needs to be created in case a change is needed. This allows for reusing a single sensor by more subscribers in case they specify the same parameters[1]. A sensor returns the same format of data as *getters* and also uses mainly the *getters* to get the data it works on. However, since the data format is the same (list of subtrees), the sensors can be chained (filtration sensors) or a sensor can use more other sensors as its input (aggregation sensor). Moreover, the

[1]However, it does not prevent optimizations like copy-on-write or similar.

simple getter (without locking) can be called to get some parameters — the following pseudocode shows an airplane sensor returning other planes within some range. Although the sensor position changes as the plane moves, the subscriber does not need to care since a new position is always used.

```
Sensor_Radar (param1: $Plane \rightarrow 1$, param2: 2000)
begin
   GET(param1, ⟨ position, x → ?1 ⟩, ⟨ position, y → ?2 ⟩, ⟨ position, z → ?3 ⟩,)
   dist := param2
   GET(Plane → ?, ⟨ position, x → $?1 - dist\ to\ ?1 + dist$ ⟩, ⟨ position,
       y → $?2 - dist\ to\ ?2 + dist$ ⟩, ⟨ position, z → $?3 - dist\ to\ ?3 + dist$ ⟩)
   filter_data()
end.
```

The first GET command gets a current position of a Plane 1 while the second uses this position to get the other planes around with their positions.

1.3.2 Actions

An action represent a world change that should be atomic and performed at one point of time. Similarly to the sensors, actions get some parameters and uses mainly the getters/setters interface. In addition, one action can invoke some other actions, in which case all of them need to be performed atomically. Each action has assigned a timestamp and a unique number — timestamps are discussed later, the unique number is propagated through the nested actions, setters, and getters to the stores, where it is used for locking. This allows us to detect which actions collide with each other and perform appropriate steps to correct that situation.

An example of an action pseudocode follows; a bus loads people at a bus stop:

```
Action_LoadPeople (ActionNo: 345, param1: $Bus \rightarrow 3$)
begin
   # get count of people in the bus
   GET&LOCK(ActionNo, ⟨ param1, PeopleCount → ?1 ⟩)
   # get a bus stop number where the bus arrived
   GET&LOCK(ActionNo, ⟨ param1, BusStop → ?2 ⟩)
   # get a capacity of the bus
   GET&LOCK(ActionNo, ⟨ param1, Capacity → ?3 ⟩)
   # get count of people on the bus stop
   GET&LOCK(ActionNo, ⟨ BusStop → ?2, PeopleCount → ?4 ⟩)

   # set new values for the bus and bus stop
   SET(ActionNo, ⟨ param1, PeopleCount → $?1 + min(?4, ?3-?1)$ ⟩)
   SET(ActionNo, ⟨ BusStop → ?2, PeopleCount → $?4 - min(?4, ?3-?1)$ ⟩)
end.
```

The first set of GET&LOCK invoke a simple getter that returns current parameter values (in case they are not locked for write, in which case the action cannot continue). Than appropriate numbers are set by setters. If the action is unable to get a lock at any time, the execution is stopped and the entity that invoked the action (*ActionCoordinator*) is informed about an action number that caused a collision. This entity can than decide whether both actions need rollback, or whether the first one can be commited (COMMIT) or whether both should be rolled back and the second one restarted.

1.4 Sensor/Action Coordinators

These serve as an interface between agents and the world.

Sensor Coordinator maintains a hierarchy of sensors created on a local machine, what are their parameters and what subscribers need which information. Therefore it can connect new subscribers to existing sensors (in case they exist already) or perform some optimizations (for example not discarding the sensor from memory if it is likely that someone will need it in a while). Every sensor perform its task only when asked by the sensor coordinator, they do not sense all the time. This allows the subscriber to specify some simulation-time dependent intervals when it needs the data, the sensing does not need to occur every *time-step* — see the time discussion below.

The sensor hierarchies are typically static; described in a separate configuration file. Every hierarchy has a name - this is an identifier agents than use while subscribing. The configuration file maps this name to a sensor, specifies its parameters and "subsensors"[2]. Although the static hierarchies would be probably sufficient for most simulations, some might want to create those at run-time. Therefore the Sensor Coordinator provides an interface for adding new sensor hierarchy at agent's request. This will be programmatic in the first release, might be changed to a simpler one in the future.

The following steps describe the "sensing workflow":

1. The subscriber tells the coordinator what data it needs (by a hierarchy name), specifies required parameters and a callback function that will be used for passing data back.
2. The sensor coordinator checks whether there is already a sensor that provides such data, if so, assigns the subscriber to this sensor together with information when (how often) the data should be passed to it.
3. If there is no such sensor, it creates new one. This might involve creating a hierarchy of sensors, in case the previous one depends on those that are not already present. There is no "subscribe" relation in between those nested sensors since the parent one always calls the "child" for data — the child does not need to know who is the parent.
4. The sensor coordinator takes care about the right moment when the data should be collected and passed to the subscribers. The coordinator calls the first sensor that than calls its children which eventually call the getters. This architecture requires all the sensors to cache the recent data with a timestamp of their validity to avoid multiple unnecessary sensing.
5. It's obvious that the sensing might need to get some data from a remote computer. To keep things simple, we allow for waiting for a response — sensor coordinator uses a thread pool (shared with action coordinator) to perform the sensing operations. On the other hand, this requires having all the sensors thread-safe. *(Alternative approach: Two-phase sensing - first identify the sensors that need update and than update them bottom-up – might be faster, all requests to remote computers might be sent at once + no thread-safety required?)*

Action Coordinator (similarly to sensor coordinator) maintains a hierarchy of actions. However, unlike the sensors, actions are designed to perform one-shot tasks, and they cannot share their children. When the coordinator is asked to execute some action, it creates a new one (with a specific parameters and unique action number); in case this action does not access setters directly and uses some other actions, they are also created. Because they cannot be shared, they doesn't need to be thread-safe.

To speed-up the action execution, even if the actions are not thread-safe, they should be reusable. In other words, action coordinator should be able to maintain a cache of actions and set different parameters to the existing action instead of creating a new one. Another optimization could be a plug-in telling the coordinator which actions are likely to collide and the coordinator can prevent their simultaneous execution (locally, or globally, which would involve network communication; however, action coordinators need to communicate with each other anyway – see Time Synchronization section).

[2]Format of this file is not determined yet

The above is true in case the actions would need to maintain some data fields. In case not, they can be reusable. Not sure about this yet.

The action coordinator also receives information about the collisions that occured during the execution and can decide what to do. Therefore, swapping the Action Coordinator module allows us to completely change the behavior from "best-effort" – let the first action continue (COMMIT) and discard/postpone the second (ROLLBACK) – to the "fair simulation" where the coordinator decides which one should be executed and in case it is the first one, rolls back both and restarts the first one.

Obviously, the actions might also need to get or set a value present on the remote host. The solution would be the same as described above for the sensors (first approach) – more thread-pool threads perform the execution and they can block waiting for some remote data/action.

1.5 Behaviors

Every agent changes the world state through a *behavior*; each agent has a separate instance. This behavior is responsible for generation of appropriate actions, which are than executed by Action Coordinator. The interface between the agent and its behavior is project-specific, for example the agent might pass a flight-plan to the behavior and the behavior than generate appropriate "move" actions. The interface realization is also not pre-defined, the agent can pass *topics* to the behavior, call methods (in which case their separation to different computers is not possible) or whatever.

First, this approach helps to separate agent logic from its physical execution, second, it allows to easily integrate non-agent entities into the world. These entities are implemented as separate behavior modules; these modules are indistinguishable from the coordinator point of view. They can also subscribe sensors; for example animals might move randomly, but run away if there is a person close to them.

2 Time Synchronization

The simulation advances in a virtual "simulation time". This time is independent of a real time - it can be synchronized (to support interactive simulations), but it is not necessary. The following text discusses an *as-fast-as-possible* mode, in which the simulation speed adjust to the hardware resources. The option where the simulation time synchronizes with the real time is described next.

Action Coordinator knows about all the *behavior*s connected to it and holds a simulation timestamp telling the point in the simulation where the behavior is ready to go; for which it has a next action ready. This does not mean that this action will be executed in the future; it would be executed in case no other previous actions will modify the sensors in the way that the agent decides (through its behavior) to do something else. This timestamp should not be set to the past, but can be set to the current time if the agent is not yet ready. In case the agent has nothing to do unless the sensors detect something, the timestamp should be set to an "infinite" value.

Each behavior is responsible for updating these timestamps in the Action Coordinator module. If the agent decides that there might come another action before the previously set timestamp, it has to update it immediately. If the action is not ready yet (requires some further computation), the timestamp should be reset to the current simulation time to stop the simulation for a while.

From the above, the action coordinator always knows about the status of each behavior, whether it is ready or not, and what is the simulation time of the next action it wants to perform. These steps describe how the simulator handles actions, sensors and their dependencies:

1. The action coordinator checkes which *behaviors* are ready, in case those that are ready specify a timestamp of a next simulation step, the coordinator asks for the actions for this next step and starts their execution.

2. After every *behavior* is ready, a minimum of timestamps is selected (globally, among all the computers). Than the appropriate *behavior*s that specified this value are queried for the actions planned for this time and these actions are executed.

3. All the behaviors are notified about the minimum timestamp value chosen. Based on this information, they can change their timestamp in the Action Coordinator (e.g. reset to this minimum to not continue until the agent takes some action, or set to the minimum plus some delay, in which case the agent needs to be aware that the simulation can continue for that delay).
4. In case there is some collision between the actions, some project-specific rules for its resolution apply - the behaviors are than notified about the actions that failed.
5. All action coordinators than notify others when all the actions are done. The simulation "current time" is set to the new minimum chosen above. All action coordinators than activate sensor coordinators.
6. Sensor coordinator checkes which subscribers require new data in this step and refreshes the appropriate sensors. New sensor data are passed to the subscribers (agents, behaviors).
7. The agents than can update their behaviors to hold a simulation, let it run or let it run to a specific point of time.
8. Sensor coordinators negotiate whether all of them finished; if so, they return control back to the action coordinators — these steps than repeat.

   Please note that since the control returns back to the action coordinator after all the sensors are processed (but probably before all the agents can react to the world change), it might be possible that simulation might advance a bit before the agents manage the behavior update; this cannot be prevented without a cooperation with the agent. However, the architecture allows for various solutions a specific project can choose from, for example:

   AgentFly currently measures the CPU usage and if it is not overloaded, it assumes that the agents are able to do all the work. This approach can be easily implemented by adding a "fake" behavior that will not pass the actions, but just set the timestamp whether it is ready or not, based on the CPU measurements.

   The CPU measures can be avoided if the agent will inform its behavior that it got a new sensor data and needs some time to compute flight-plan updates. The behavior can set a timestamp to a current time or a bit greater value, therefore effectively pause the simulation (immediately or after a while).

   I-Globe uses another method, which fit nicely to this architecture. After each simulation step, all the agents need to tell that they are done with their work for this "round". In this case, they will tell their behaviors that they can advance a "ready" timestamp that was previously reset after a minimum value was chosen.

2.1 Synchronization with Real Time

By the synchronization with real time we mean that we define that there should be a delay of exactly t milliseconds between the two simulation "ticks". For this, we need a generator of these ticks — this generator should be synchronized globally over all the computers.

This generator sends the ticks to the Action Coordinator — until the t is large enough that all the components cope with their work, than the coordinator simply waits for the next tick before it passes control to the sensor coordinator. However, if t is small and the tick comes before all the behaviors are ready, two options are possible:

- The action coordinator **will wait** for the remaining behaviors to become ready. In such a case, the simulation might be a bit snatchy but hopefully the total time of the simulation will be fine, since the delay might be smaller in the next round (the ticks generator is not affected, it just produces ticks by the same speed).

- The action coordinator **will not wait** for the remaining behaviors; they need to be ready for this situation and be able to provide some alternative action instead the correct one (or none if it is not necessary). The action coordinator will ask the behaviors for these alternative actions and execute them.

  > This approach is needed by the AgentFly simulator, where the aeroplanes are flying based on their old flight-plan in case the agent was not able to provide a new one.