SUBPART SEVEN

Within-Day Replanning

CHAPTER 30

Within-Day Replanning

Christoph Dobler and Kai Nagel

30.1 Basic Information

30.1.1 Implementation Alternative 1

Entry point to documentation:

`http://matsim.org/extensions` → withinday

Invoking the module:

`http://matsim.org/javadoc` → tutorial → `RunWithinDayExample` class

Selected publications:

See Section 30.4.2.

30.1.2 Implementation Alternative 2

Entry point to documentation:

`http://matsim.org/extensions` → withinday

Invoking the module:

`http://matsim.org/javadoc` → tutorial → `RunOwnMobsimAgentUsingRouter` class

Selected publications:

See Section 30.4.3.

How to cite this book chapter:

Dobler, C and Nagel, K. 2016. Within-Day Replanning. In: Horni, A, Nagel, K and Axhausen, K W. (eds.) *The Multi-Agent Transport Simulation MATSim*, Pp. 187–200. London: Ubiquity Press. DOI: http://dx.doi.org/10.5334/baw.30.

30.2 Introduction

In recent years, transport planning and traffic management interest in unforeseeable, or only partially foreseeable events within scenarios has increased. Partially foreseeable events often occur with taxis and car sharing. For example, agents with a planned taxi trip cannot know in advance which taxi will be available when they need one. When using car sharing, an agent might walk to the car sharing station and check whether a car is available or not. If it is not, the agent could either decide to wait, or change its plan and switch to another transportation mode. Road accidents, terrorist attacks or disasters such as earthquakes are examples of completely unpredictable events.

As discussed earlier, traditional simulation approaches (used in default-MATSim) calculate demand-supply equilibria using an iterative process. There, it is assumed that a typical situation is simulated where agents can rely on their experience from comparable situations, like previous iterations. Applying an iterative approach to a scenario with unexpected events results in problems like illogical agent behavior, producing false results. In the next section, these problems, as well as an alternative simulation approach, are presented. On one hand, this approach—called within-day replanning—simulates only a single iteration, avoiding problems resulting from an iterative simulation process. On the other hand, this approach does require a more detailed behavioral model for the agents. Subsequently, using MATSim as a base, the iterative approach is discussed, followed by two different implementations of the within-day replanning approach into the framework, including discussions of the technical implementations.

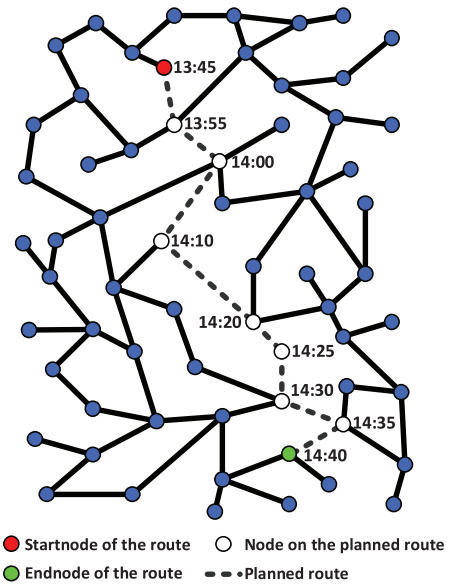
30.3 Simulation Approaches

30.3.1 Iterative Simulation Approaches

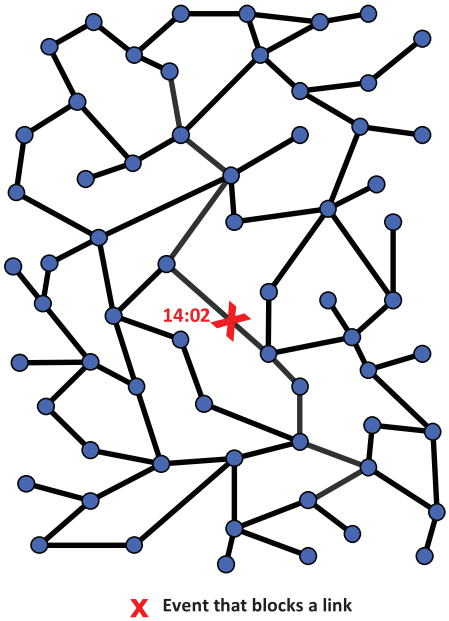
An iterative day-to-day replanning approach is appropriate as long as the scenario describes a *typical* situation or day. For such scenarios, it is feasible to assume that agents are familiar with typically occurring events like traffic jams during peak hours. Therefore, they try to avoid driving during those times, or use alternative routes with less traffic. However, if the scenario contains unexpected events that the agents cannot foresee, e.g., accidents or heavy weather conditions, using an iterative approach is not an appropriate choice. First, a user equilibrium will not be reached in such a scenario because agents do not have enough information to choose optimal routes and daily activity plans. Another problem is the optimization process itself. Even if an agent chooses its routes randomly due to a lack of information, it will eventually find a good route if it tries enough different routes.

Figure 30.1 shows a simple example scenario where an iterative approach would produce illogical and faulty results. In Figure 30.1(a), an agent's planned route in a sample network is shown, including the times when the driver passes each node of the route. Clearly, those times are only valid if no exceptional event occurs. Figure 30.1(b) shows a link where an event, like an accident, blocks that link for two hours. As a result, the agent reaches its destination two hours later than expected (Figure 30.1(c)). When this scenario is iterated, the agent recognizes that its route has a much higher travel time than expected and therefore it will choose another route. The traffic jam caused by the accident will probably also increase travel times on links next to the blocked link. Therefore, the agent might find a route which is quite different than the original one (Figure 30.1(d)). A closer look at the node where the new route deviates for the first time from the original one shows that this occurs even before the accident happened, which is unfeasible and illogical.

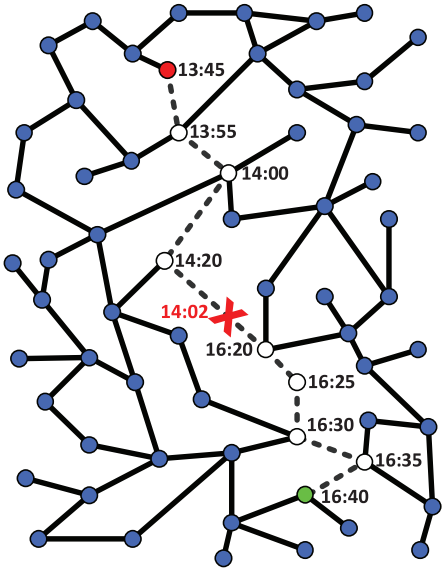
An obvious solution to avoiding such problems is using an alternative simulation approach without an iterative optimization process. The next section discusses such an approach and the requirements that must be fulfilled.



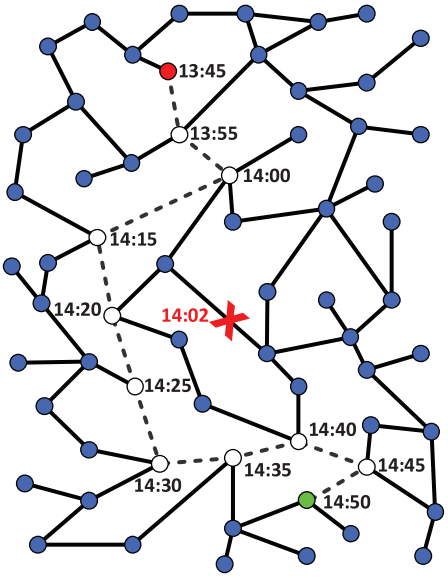
(a) Network with planned route.



(b) Network with exceptional event.



(c) Network with exceptional event and planned route.



(d) Network with exceptional event and adapted route.

Figure 30.1: Exceptional event in a network.

30.3.2 Within-Day Replanning Approach

A within-day replanning approach uses a significantly different strategy from that of an iterative approach. Instead of multiple iterations, only a single one is simulated. Thus, it is now essential that agents can adapt their plans during this iteration without having information from previous

iterations available. To do so, they have to continuously collect information and take into account their desires, beliefs and intentions when they decide how to (re)act.

While iterative approaches can use best-response modules, a within-day approach has to use something that might be called a best-guess module. Travel times are an obvious example. In an iterative approach, travel times can be collected from the previous iteration or even be averaged over several past iterations. The nearer a stable system is to a relaxed state, the smaller the differences in travel times between two iterations. This is not possible in a within-day approach. Even if an agent has perfect knowledge, it can only assume how the traffic flows will evolve in the future. To do so, it can take different information into account to estimate travel times. It could, for example, take travel times from a typical day without exceptional events and combine them with information it gathers during the simulated day. Depending on the amount and the quality of this information, the agent might rely more or less on its experience.

Therefore, the decision-making process of an agent becomes an important topic. In an iterative approach, each agent has total information and can thus select the best route. Due to limited available information, this is not possible in a within-day approach. One agent could, for example, choose a route where expected travel time is very short, but also very uncertain. Another agent might not be willing to take that risk and therefore select a longer route where the assumed travel time is more reliable. Perception of information might also vary between agents; one could rely on media traffic information, another might ignore it.

Each within-day replanning action is categorized by two parameters—the replanned element of the plan (an activity or a trip) and the point in time when the replanned plan element is executed (right now or at a future point in time). If an activity is replanned, several changes are possible. Its start and end time can be adapted, its location can be changed, it can be dropped, or created new from scratch. For a trip, origin and destination, route, mode of transport and departure time can be replanned. Often replanning one single plan element results in a chain reaction that forces replanning of other plan elements. If, for example, an activity is dropped, the trips from and to this activity have to be merged.

The second parameter categorizing a replanning action depends on when the replanned plan element is executed. This could be either the currently performed plan element or one being performed in the future. Clearly, in a currently performed plan element, not all previously mentioned replanning actions could be conducted, e.g., start time of an activity or transport mode of a trip currently being performed can no longer be adapted.

Due to the limited available information, a within-day replanning approach will, in contrast to an iterative approach, not converge to a user equilibrium. Decisions made during the simulated time period may seem to be optimal when they are made. However, evaluated retrospectively, an agent might realize that they were not.

Figure 30.2 shows how within-day replanning can be integrated into MATSim's iterative optimization loop. An additional block builds another (inner) loop with the mobility simulation. Depending on the type of simulated scenario, the outer loop can be skipped.

30.3.3 Combined Approaches

An alternative to iterative, or within-day replanning only approaches, is to combine them. An obvious application is solving situations that cannot be planned exactly in advance, like parking or car sharing. An agent is, for example, able to plan a parking activity, but it cannot anticipate which parking spots will be available when the agent arrives. Thus, within-day replanning can be used when the agent starts its parking choice.

Other agents might want to share their cars, so an actual meeting must be confirmed. This can be ensured using within-day replanning. If the driver arrives too early, a *waiting* activity is added to its plan; otherwise the agent being picked up will perform a *waiting* activity until the car arrives.

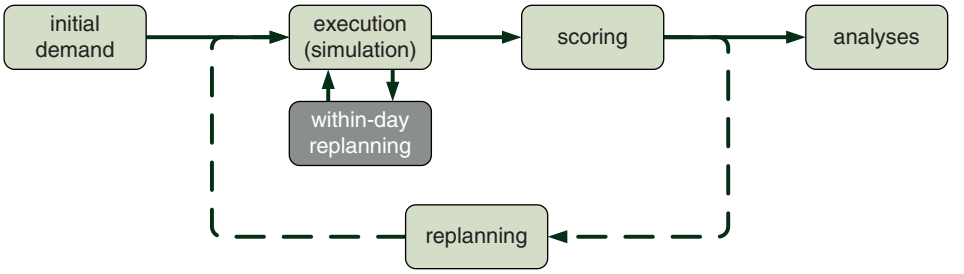


Figure 30.2: (Iterative) within-day replanning MATSim loop.

30.4 Implementation

30.4.1 General Thoughts

Within-day or en-route replanning means that travelers replan during the day or while they are on their route. This means that the simulation needs to find some way to influence the agent while the mobsim (network loading) is running. For the MATSim main network loading module, the so-called QSim, this could be achieved by inserting an agent-loop, as follows:

```
void doSimStep() {
   for ( each agent ) {   // <-- agent loop
      agent.doSimStep() ;
   }
   for ( each link ) {
      link.doSimStep() ;
   }
   for ( each node ) {
      node.doSimStep() ;
   }
}
```

In this loop, each agent has the chance to deliberate in every time step. Clearly, the agent can decide that he/she has nothing to deliberate and return immediately.

Such an approach does, however, lead to computational challenges. Going through all links and nodes in every time step is already an expensive operation and a number of efficiency improvements (such as "switching off non-active links") are contained in the code. Also, the number of links or nodes is typically an order of magnitude smaller than the number of synthetic persons in a scenario. Thus, some massive optimization would have to be undertaken in order to make the above approach computationally efficient.

An alternative approach to the above is to ask each agent only when a decision needs to be made. The most important decision for a driver is to chose the next link, i.e.,

```
class MyDriverAgent implements DriverAgent {
   ...
   @Override
   public Id<Link> chooseNextLink() {
      <algorithm to determine ID of next link>
      return nextLinkId ;
   }
}
```

Similar implementations are needed for all other queries that could be asked of the agent, for example:

- Should the trip end on the current link?
- Should the agent get off at the current stop?
- What is the `ID` of the vehicle to be used for a trip?

From the agent's perspective, such an approach might be called *event driven*, since the agent performs only mental activity at such events.

There is, indeed, a mechanism to program such agents and to insert them into the QSim. This is discussed in more detail in Section 30.4.3.

A challenge inherent in that approach is that the complete agent needs to be re-programmed. This agent needs to have enough capabilities to be oriented about itself; for example, it needs to be able to compute plausible routes.

On the other hand, there are situations where the capability to decide the turn at each intersection while en-route is, in fact, not needed. For example, for typical evacuation applications, it makes sense to start all agents on their normal daily plans. When an emergency warning is distributed, the simulation can go once through all agents and decide how they react. This will be done by replacing some, or all, future elements of the current plan. In some applications, this may happen more than once; for example, if recommended evacuation directions change because of a shift in the wind. In other applications, evacuating agents could become stuck in unexpected congestion which might trigger en-route re-routing. This may, however, be restricted to relatively small regions, and it may be sufficient to go through such a replanning loop, perhaps every 300 simulated seconds.

For such applications, the plan-based approach (Section 30.4.2) is more suitable. Rather than having each agent answering certain queries in every time step or at every intersection, the plan-based approach first waits for a trigger (such as an emergency warning, or unexpected congestion), then decides on the affected agents, then goes through those agents and changes the future part of their plans. This is not only conceptually easier than having every agent answer for him-/herself, but it is also computationally more efficient, since it is only called when it is triggered and impacts only the affected agents.

Overall, implementers and users will have to balance their needs. If there are relatively few times when agents should re-plan, and these times can be easily identified by, i.e., corresponding to an emergency signal, then this is an indicator for the plan-based approach. If, on the other hand, an agent goes into the simulation mostly or entirely without a plan, like an entirely reactive taxi driver, then this speaks for replacing the agent.

MATSim provides infrastructure for both approaches. The plan-based approach currently provides more support infrastructure, i.e., many important use cases can be implemented by re-using existing methods. The approach that replaces the agent, in contrast, provides more flexibility. In particular, it allows agents to make decisions at the latest possible time without additional computational overhead. While this is not entirely realistic behaviorally, such an approach is often desirable from a simulation perspective, where one does not want reproducibility of simulations depend on, e.g., random elements such as how far an agent plans ahead.

30.4.2 Implementation Alternative 1: Plan-Based Implementation

When adding within-day replanning to MATSim, its iterative loop (see Figure 1.1) has to be adapted as shown in Figure 30.2. On one hand, the additional *within-day replanning* module is added, which interacts with the mobsim. On the other hand, multiple iterations are only necessary if a combined simulation approach is used.

The implementation is realized as so-called `MobsimEngine` which can be plugged into the QSim. In every simulated time step, the QSim iterates over all registered `MobsimEngines` and allows them to simulate the current time step. Besides simulation of the traffic flows, those engines are also able to let agents start or end activities. The engine containing the within-day replanning logic (called `WithinDayEngine`) does not simulate traffic flows, but tracks agents and adapts their plans. Doing so is separated into two steps. First, agents whose plans have to be adapted in the current time step are identified. In a second step, the adaption of their plans is performed.

Figure 30.3 shows the structure of the `WithinDayEngine`. Multiple `Replanners` can be registered to the engine. Each `Replanner` represents a unique replanning strategy like re-routing or time mutation and uses a set of `AgentSelectors` that communicate with agents and select those who are given the opportunity to adapt their plans. An `AgentSelector` can be seen as an information-distributing unit, like a radio station or a policeman. Therefore, not every *AgentSelector* communicates with all agents. For example, agents at home will probably listen to the radio, but agents walking in the park will not. Each `AgentSelector` returns a list of agents to its superior `Replanner`, which then adapts those agents' plans.

Responsibilities are divided between `Replanners` and `AgentSelectors`. The first ones are responsible for adapting the agents' plans, but they should not check whether an agent should be replanned or not. If, for example, a `Replanner` updates an agent's route, it has to be ensured by the `AgentSelectors` that only agents who are currently performing a leg are replanned. In turn, `AgentSelectors` should select agents who have to be replanned but should not change their plans. As a result of this division, the often time-consuming replanning of the agents' plans can be performed using parallel threads, which leads to an almost linear speed-up. In general, simulation results do not depend on the order in which agents are replanned. *Replanners* which use random

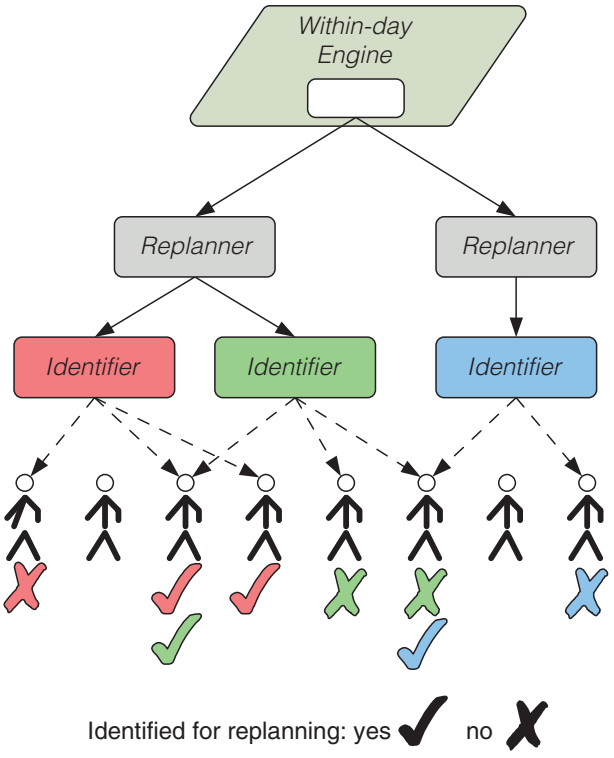


Figure 30.3: WithinDayEngine.

numbers are a special case. In the present implementation, their *random number generator* is re-initialized for every replanned agent, using a deterministic value (e.g., a combination of the agent's ID and the current time step). On one hand, this ensures that an agent's decisions can be reproduced even when the global sequence of random numbers changes. On the other hand, the simulation outcomes do not change if the number of threads used for the replanning is changed.

Running the AgentSelector(s) to select those agents who have to adapt their plans is performed sequentially. On one hand, an AgentSelector's runtime is typically very short and therefore no significant performance losses are expected. On the other hand, this makes the design robust so it cannot produce race conditions which could occur if multiple instances of an AgentSelector run concurrently. An example would be an AgentSelector, which selects agents on household level, i.e., if a member of a household is identified, also all other members are added to the list of agents who have to be replanned. In an approach with parallel running instances of an AgentSelector, an instance could identify member "A" of a household while concurrently another instance could identify member "B" of the same household. As a result, the household's members would be duplicated in the list of agents to be replanned—once added by each AgentSelector instance.

Replanner implementations are available for any basic change of an agent's scheduled daily plan. All trips and activities can be adapted, although some replanning operations are not available when trip or activity has already been started. Possible adaptations are:

- current trip (route, destination),
- future trip (add, remove, mode, route, origin, destination),
- current activity (end time), or
- future activity (add, remove, location, type, start and end time).

For complex plan adaptations, those basic Replanners can be combined. If, for example, an agent currently performing a trip changes the destination of its next activity, routes of the current and next trip must be adapted.

Additionally, four basic AgentSelectors have been implemented so far. They identify agents, which are...

- performing an activity,
- performing an activity which will end in the current time step,
- performing a trip, or
- performing a trip and are going to move to another link.

Often, only a subset of the population, e.g., only male agents, or agents currently traveling in a car, needs to be identified. To prevent that the same functionality having to be implemented multiple times, so-called AgentFilters are introduced. Their task is to remove agents not meeting the filter criteria from an agent set. Using AgentFilters not only avoids duplicated code, but can also reduce computation effort: for example, two AgentSelectors which should identify only agents currently traveling in a certain part of the network. Without AgentFilters, each of them would have to track all traveling agents and their current positions. When this functionality is moved to an AgentFilter, the two AgentSelectors can share a single instance of that filter.

Basically, simple and re-usable functionality should be implemented as AgentFilters, while more complex and/or decision-making functionality should be part of an AgentSelector. Again, an example: e.g., a scenario modeling the search for a parking space: a filter can be utilized to take only agents currently traveling by car into account. The AgentSelector solves the more complex tasks, such as deciding when the agent starts its search, or selecting the searching strategy to be applied.

Three basic `AgentFilters` have been implemented so far. They filter agents which are not...

- part of a predefined agent set,
- currently using a transport mode included in a given set, or
- currently located on a link included in a predefined set.

In addition to the logic identifying agents and adapting their plans, another important within-day replanning framework component is code that continuously collects information and provides it to the `AgentSelectors`. These decide,based on that data, whether agents are replanned or not. In a time step-based approach—as realized by the QSim—collecting, analyzing and aggregating data, as well as providing it, can be easily realized. Figure 30.4 shows the structure of a QSim's time step. Each time step is separated into three phases:

Phase 1:

```
before time step
```

Phase 2:

```
do sim step
```

Phase 3:

```
after time step
```

During phase 2 all registered `MobsimEngines` simulate the current time step. Phases 1 and 3 allow code execution before or after simulation of the current time step. A class can collect data such as link travel times during phase 2. phase. Then, the collected data can be analyzed and aggregated in phase 3. In the next time step, the `WithinDayEngine`'s `AgentSelectors` can use that data for their decisions. The `WithinDayEngine` is always the first `MobsimEngine` executing its `doSimStep` method, ensuring that no agent has changed its status since phase 3 of the previous time step. As a result, the `AgentSelectors` make their decisions on current data.

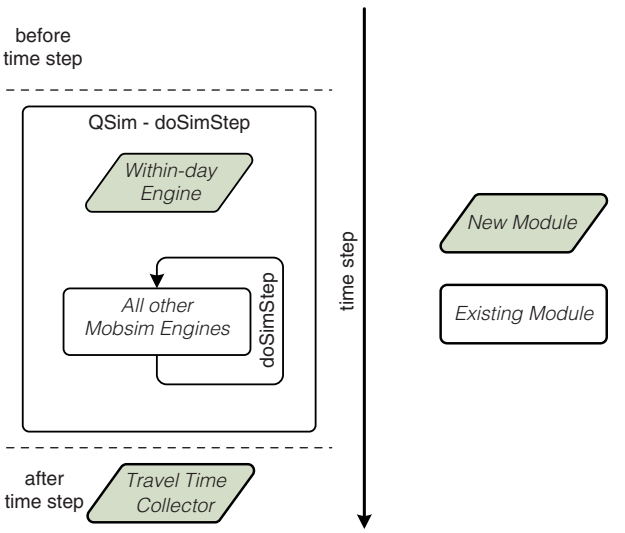


Figure 30.4: QSim time step.

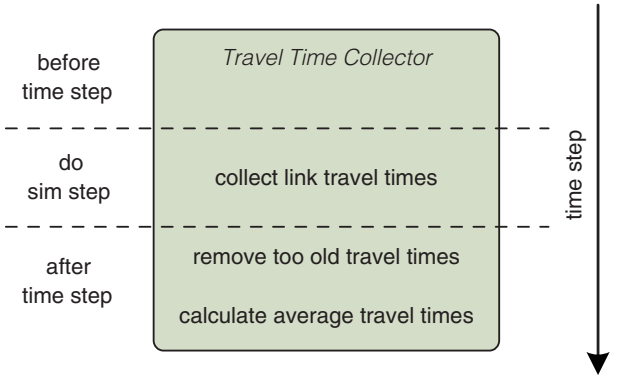


Figure 30.5: TravelTimeCollector.

An example of this type of class is the so-called `TravelTimeCollector`. It provides actual link travel times to the `Replanners` by collecting and averaging travel times of agents that have recently passed a link during a given time. A typical time span is 15 minutes; older link travel times are ignored. Specific time span duration has an important impact on travel times reported to the `Replanners`. On one hand, significant changes in link travel times will be communicated very slowly, if the time span is too long. On the other hand, a too short duration will overrate outliers.

The `TravelTimeCollector` is a simple, but efficient, implementation of a within-day travel time calculator. It does not incorporate features like traffic flow predictions or dynamic recent travel times weighting based on historic data. Because it does not factor in such features, it is very robust, even in scenarios where traffic flow conditions change dramatically.

The current MATSim code differentiates between `Person` and `MobsimAgent`. `Person` can be seen as a very simple Q-learning entity, possessing multiple `Plans` ("actions"), each with an expected score updated with every plan run. Thus, a `Person` is consistent over the iterations; in fact, the internal state of each `Person` is written to file at the end of the iterations. `MobsimAgent`, in contrast, is instantiated every time the QSim is called, and does not exist beyond the QSim running time. A `MobsimAgent` is essentially reactive, queried by the framework about decisions when approaching intersections, arrival points, or public transit stops. In the standard implementation, these queries are answered by the plan, but other implementations can be used and/or additional `MobsimAgents` can be added which do not correspond to `Persons`.

This leads to a question; should within-day adaptations to the `Plan` be passed through to the `Person`? Let us call the actual trajectory through the system the "executed plan". This can be different from the original plan, i.e., a different route, different departure times, different modes, etc. The original plan cannot just be replaced by the executed plan, since it is not clear that the executed plan, when used as input, will have itself as expected output. In consequence, it is not possible to treat the executed plan together with the just-obtained score as an action-value pair in the sense of Q-learning, since the score was obtained from the *original* plan, not from the executed plan.

As a result, the code uses a copy of the original plan and modifies the copy. The score, however, is given to the original plan. The implementation is able to *also* memorize the executed plan and add it to the set of plans. This functionality, however, is experimental.

In certain situations, setting the original to the executed plan clearly does not make sense; a parking search is one (Waraich et al., 2013c, 2012).

A person's plan contains, as destination, the location where a free parking space is expected. However, if the agent realizes in the mobility simulation that there is no free space left, it starts looking for a free parking spot. As a result, the agent's route is extended. This extension has to

be local in the agent's route, since it is only necessary in the current iteration and probably not in another one, where the initially selected parking spot is available.

Capabilities of this within-day replanning implementation are shown and discussed by Dobler (2013), based on two sets of experiments. The first set is based on a model of Zürich city, where it is assumed that capacities of several city-center arterial roads are drastically reduced during the morning peak. Traveling agents are given the opportunity to bypass the resulting traffic jams by adapting their routes, using within-day replanning. As a result, average travel time of an agent affected by the incident is reduced from 42 to 23 minutes. Also interesting is that even if only 50 % of the population adapts its routes, average travel times are reduced to 25 minutes.

The second set of experiments uses within-day replanning to create agents' initial routes. The results are compared to runs where routes are created before the simulation starts, without traffic flow information. Results indicate that agents' average travel times are already very close to the values in a relaxed state. When using MATSim's traditional approach, 10 to 15 iterations must be performed before average travel times reach this level.

30.4.3 Implementation Alternative 2: Replacing the Agent

According to Russel and Norvig (2010), an agent is "anything that can be viewed as perceiving its environment through sensors and acting upon that environment through effectors." As stated above, MATSim has agents on two levels:

- `Person` is a Q-learning agent that is persistent over the iterations.
- `MobsimAgent` is a reactive agent that only exists during the mobsim.

For the Q-learning agent, perception works through the events; i.e., events are used to compute the score, build mental models to generate alternatives, etc. Acting on the environment works through plan selection.

For the reactive agent, perception works more directly through callback methods, such as the simulation notifying the agent it has just moved through an intersection. Acting on the environment works through making decisions at decision points, e.g., about turning directions at intersections, or whether to board a certain bus.

As discussed, the approach described in Section 30.4.2 assumes that the reactive agent still has followed (and generally follows) a plan. There may, however, be situations where this is inappropriate: for example, when the agent makes up the route as it goes, or when one wants to investigate models where each agent has its own perception and deliberation, rather than some external algorithm modifying its plan. As also mentioned earlier, there is no clear rule governing when and where an approach is better; it depends both on both project requirements and on the developer's personal preferences. Here, with this in mind, we will look at `MobsimAgents` that no longer have a pre-computed plan, but make decisions as they go. There is also a class `DynAgent`, which wraps around `MobsimAgent`, making it easier to use and providing additional infrastructure (Section 23.4).

30.4.3.1 Agent Interface

The `DriverAgent` interface structurally looks as follows:

- `Id chooseNextLinkId()`—agent is asked at intersections and needs to return how to proceed.
- `boolean isWantingToArriveOnCurrentLink()`—agent is asked if it wants to arrive on the current link.
- `void notifyMoveOverNode(Id newLinkId)`—agent is notified that it has traversed the intersection and entered a new link.

The rest comprises relatively simple bookkeeping methods like `getId()`—the agent needs to know its own identifier.

If it is assumed that the agent does not only replan en-route, but also while at activities, then the `MobsimAgent` interface also must be implemented. This is a bit more involved; important methods are:

- `endLegAndComputeNextState(...)`—agent is notified that the current transport leg has ended, and the agent internally needs to decide how to continue.
- `endActivityAndComputeNextState(...)`—agent is notified that current activity has ended; the agent internally needs to decide how to continue.
- `setStateToAbort(...)`—if a leg or an activity was not ended cleanly: this could happen if `chooseNextLinkId()` returns a link that is not outgoing from the current node.[1]
- `getState()`—agent needs to return its current state, which essentially either returns `ACTIVITY` or `LEG`; most important here is that the framework obtains information about whether the agent wants to start a new activity or leg.

Again, everything else concerns bookkeeping methods.

30.4.3.2 Agent Insertion

The code accepts several ways to insert such a self-programmed `MobsimAgent` into the code, but the preferred method is using the `AgentSource` interface, as follows:[2]

```
class MyAgentSource implements AgentSource {
   // constructor
   MyAgentSource ( Guidance guidance ) {
      ...
   }
   public void insertAgentsIntoMobsim() {
      // insert agent:
      MobsimAgent ag = new MyMobsimAgent( guidance ) ;
      qsim.insertAgentIntoMobsim(ag) ;

      // insert vehicle:
      // ...
      qsim.createAndParkVehicleOnLink(veh, linkId );
   }
}
```

`Guidance` helps the agent with making decisions, see below.

30.4.3.3 Perception, Decision, Integration

The agents somehow need to perceive their environment. The simulation tells the agent where it is, via `notifyMoveOverNode(Id<Link> nextLinkId)`. In general, however, this will not be sufficient. For example, the agent may want to be informed about congestion, or evacuation directions.

A general way to achieve this is to use the `Events` channel.

We would probably suggest separating observer, guidance, and the agent itself.

---

[1] Despite the name of the method, the agent can recover.

[2] See `http://matsim.org/javadoc` → main distribution → the `AgentSource` class for a pointer to a working code example.

Observer The observer would probably listen to events:

```
class MyObserver implements BasicEventHandler {
   @Override
   public void handleEvent(Event event) {
      ... // memorize information
   }
   ...
}
```

For working code, see `http://matsim.org/javadoc` → main distribution → `RunOwnMobsimAgentWithPerception` class and related.

Guidance A guidance object might give advice to agents. It could, for example, be designed as follows:

```
class MyGuidance {
   MyGuidance( MyObserver observer ) {
      ...
   }
   Id<Link> chooseNextLinkId( Id<Link> currentLinkId ) {
      ... // compute and return decision
   }
}
```

For working code, see `http://matsim.org/javadoc` → main distribution → `RunOwnMobsimAgentWithPerception` class and related.

Agent The agent needs access to the guidance object:

```
class MyAgent implements MobsimDriverAgent {
   MyGuidance guidance ;
   MyAgent( MyGuidance guidance ) {
      this.guidance = guidance ;
   }
   ...
   @Override
   Id<Link> chooseNextLinkId() {
      return this.guidance.chooseNextLinkId( this.currentLinkId ) ;
   }
   ...
}
```

For working code, see `http://matsim.org/javadoc` → main distribution → `RunOwnMobsimAgentWithPerception` class and related.

Control script This would be plugged together by a variant of the following script:

```
Controler ctrl = ... ;
...
// create observer object:
MyObserver observer = new MyObserver() ;
// add into events channel:
ctrl.addEventsHandler(observer) ;
// create guidance object:
MyGuidance guidance = new MyGuidance( observer ) ;
// create mobsim factory and set into controler:
ctrl.setMobsimFactory(new MobsimFactory(){
   public Mobsim createMobsim(Scenario sc, EventsManager ev ) {
      MobsimFactory factory = new QSimFactory() ;
```

```
      QSim qsim = (QSim) factory.createMobsim(sc, ev) ;
      // add agent source into mobsim:
      qsim.addAgentSource( new MyAgentSource( guidance ) ) ;
      return qsim ;
   }
}) ;
...
ctrl.run() ;
```

The above "script" uses an anonymous class for the `MobsimFactory`. This method of writing code is quite convenient for adapting MATSim to individual needs, also see Chapter 45.

For working code, see `http://matsim.org/javadoc` → main distribution → `RunOwnMobsimAgentUsingRouter` class and related.

30.4.3.4 `DynAgent`

As stated earlier, there is also a class `DynAgent`. It wraps around `MobsimAgent`, making it easier to use and providing additional infrastructure (Section 23.4).