JÜLICH FORSCHUNGSZENTRUM

CNS 2013 tutorial *developing neuron and synapse models for NEST*

Part II: scheduling and the model API in NEST

July 13, 2013 | Jochen Martin Eppler (j.eppler@fz-juelich.de)

Institute for Neuroscience and Medicine (INM-6)
Computational and Systems Neuroscience

Outline

Scheduling in the NEST kernel

The API for neuron models

The API for synapse models

Parallelization issues

Model development workflow

References and further reading

Definitions and concepts

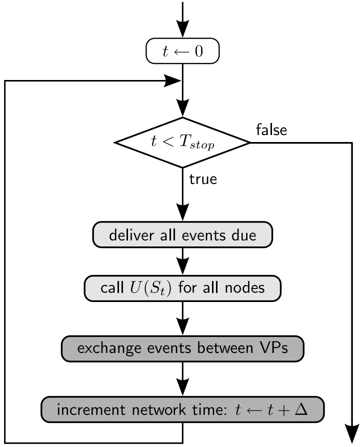
- The network in NEST is a directed, weighted graph
  - *Nodes* represent either *neurons* or *devices*
  - *Edges* represent *synapses* between nodes

- Nodes are updated on a fixed-time grid, while spikes can be on the grid or in continuous time
- Neurons can be arbitrarily complex, not just point neurons

- Synapses are updated in an event-driven fashion

- Parallelization and inter-process communication is handled transparently by NEST

Simulation loop



$t \leftarrow 0$
$t < T_{stop}$
false
true
deliver all events due
call $U(S_t)$ for all nodes
exchange events between VPs
increment network time: $t \leftarrow t + \Delta$

- Simulation starts at $t = 0$
- We simulate for T_{stop} ms
- $U(S_t)$ propagates the neuron state S to time t
- VPs are virtual processes
- Δ is the minimal delay in the network

□ parallel on all threads
▨ parallel on all processes

Network update

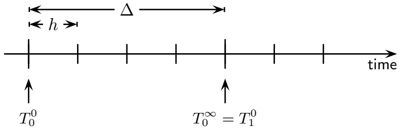
- Neurons and devices are updated in the order of their creation

- During the run of the update function, all previous events are taken care of, and new events are created
- Spikes are buffered for local and remote delivery in the next time slice
- All other events are delivered immediately to local nodes

- Devices for stimulation and recording are replicated on each VP, which also deliver locally

Node update

During an interval of the minimal transmission delay in the network (Δ), neurons are effectively decoupled.



- The update function of nodes (U) is called every Δ steps
- The nth time slice of length Δ starts at $T_n^0 = n \cdot \Delta$ and ends at $T_n^\infty = (n+1) \cdot \Delta$
- Internally, nodes use a time step of h (e.g. for solvers)

Events

Each neuron type sends exactly one type of event, but can receive many different types.

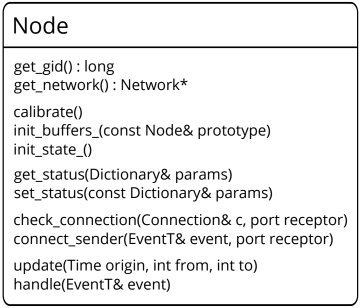
- **SpikeEvent** is the most common event type for neurons. It transmits one or more spikes (*multitude*)
- **CurrentEvent** is sent by stimulating devices to transport a current value
- **DataLoggingRequest** is sent by the *multimeter* to query recordables. Data is stored in the neuron

Common infrastructure

- **Ring buffers** are used for buffering incoming spike and current events
- **Network::send()** is a convenient wrapper for the internal event delivery infrastructure
- The **Time class** allows an easy conversion between ms and integer simulation time steps
- A **Testsuite framework** allows the integration of own unit tests into NEST's battery of tests
- The **Name class** ensures consistent naming of public variables and quantities in model classes

The model API

Neuron and device models are derived from the base class Node.



Node

```
get_gid() : long
get_network() : Network*

calibrate()
init_buffers_(const Node& prototype)
init_state_()

get_status(Dictionary& params)
set_status(const Dictionary& params)

check_connection(Connection& c, port receptor)
connect_sender(EventT& event, port receptor)

update(Time origin, int from, int to)
handle(EventT& event)
```

- Each node has a unique id
- Through **get_network()**, nodes access the Network
- The user interface consists of only two functions
- A connection handshake ensures valid connections

(Functions with EventT as argument type exist once for each event type)

Calibration and initialization

Node::init_state_(const Node& prototype)

- Called upon **ResetNetwork**
- (Re-)initialize the model with the state of *prototype*

Node::init_buffers_()

- Called before simulation, if *buffers_initialized* is false
- Empty ring buffers and other buffers

Node::calibrate()

- Called before simulation
- Set variables depending on the simulator state (e.g. h)

Calibration and initialization - example

models/iaf_neuron.cpp, lines 177-232

Update

Node::update(Time origin, long from, long to)

- Propagate to the end of the time slice starting at *origin*
- Or: *from* and *to* allow simulating fraction of the slice

- This function is called always at the beginning of the slice

- During update, events may be created and sent
- Internally, the resolution should be h (e.g. using a loop)
- The model may be arbitrarily complex (or simple)

Update - example

models/iaf_neuron.cpp, lines 238-281

Receiving events

Node::handle(EventT& event)

- **handle()** is called for each incoming event
- The connection handshake ensures that only valid connections can be created

- Ring buffers can be used to store incoming events
- **Event::get_rel_delivery_steps(Time origin)** can be used to get the *lag* inside the time slice
- Events are handled during the next **update()** cycle

Receiving events - example

models/iaf_neuron.cpp, lines 283-306

models/iaf_neuron.h, lines 244-245

Structured data storage

The different categories of variables are stored in structs to increase the readability of the model.

- **Parameters_** are variables, which are set by the user, but don't change dynamically during simulation
- **State_** comprises the (observable) dynamical variables of the model
- **Buffers_** are data structures for temporary storage of data, e.g. file handles or buffers for incoming spikes/currents
- **Variables_** collects all remaining (helper) variables needed for the implementation of the model

Structured data storage - example

models/iaf_neuron.h, lines 172-271

Parameters and state variables

The values in the structs for parameter and state must be set and read by the user.

- **GetStatus** and **SetStatus** allow the user to change/see the current values of variables in the model

- *Parameters_* and *State_* provide getters/setters
- Getters/setters are called on temporary objects to guarantee consistent values in case of errors
- *Dictionaries* are used to transfer data back and forth between user and simulation kernel
- Standard *names* are used to guarantee consistency

Parameters and state variables - example

models/iaf_neuron.cpp, lines 57-142

models/iaf_neuron.h, lines 335-361

Recordables

To allow recording of data at runtime, a generic interface for the multimeter is available.

- Recordable analog quantities are inserted into a table that maps variable names to functions
- Data collection and buffering is carried out by the function **record_data()** of the UniversalDataLogger
- Recordable variables can be queried using **GetStatus**
- The multimeter can be configured to record to screen, file, or memory

Recordables - example

models/iaf_neuron.h, lines 164-165

models/iaf_neuron.h, line 276

models/iaf_neuron.h, line 295

models/iaf_neuron.cpp, line 39

models/iaf_neuron.cpp, lines 45-50

models/iaf_neuron.cpp, lines 297

Instantiation of nodes

All nodes are created by a factory, which copies the prototype node for that type.

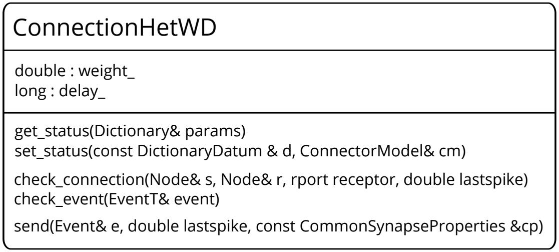
- Model implementations need to define a copy constructor, which copies *parameters*, *state*, and *buffers* of the prototype

On the user interface side of things:

- New prototypes can be created using **CopyModel**
- Model parameters are set with **SetDefaults**
- Nodes are created using the command **Create**

The synapse model API

Synapse models are derived from the base class Connection or ConnectionHetWD.



ConnectionHetWD

double : weight_
long : delay_

get_status(Dictionary& params)
set_status(const DictionaryDatum & d, ConnectorModel& cm)
check_connection(Node& s, Node& r, rport receptor, double lastspike)
check_event(EventT& event)
send(Event& e, double lastspike, const CommonSynapseProperties &cp)

(Functions with EventT as argument type exist once for each event type)

- Plastic synapses can modify the weight in **send()**

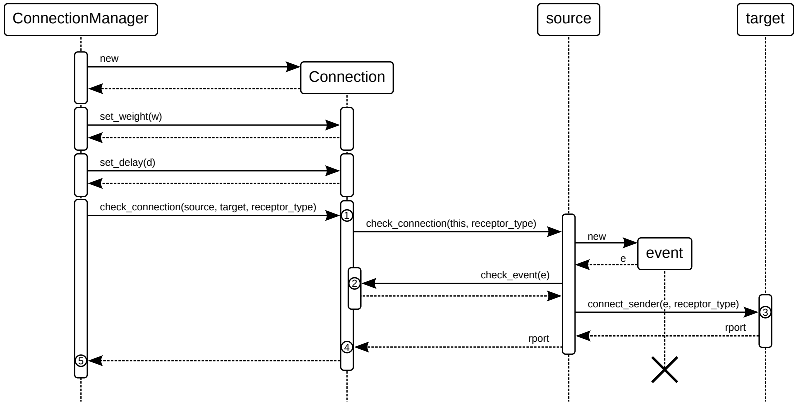
Synapse models - example

models/stdp_connection.h, lines 175-187

models/stdp_connection.h, lines 195-237

models/stdp_connection.cpp, lines 57-77

Connection handshake



ConnectionManager
source
target
new
Connection
set_weight(w)
set_delay(d)
check_connection(source, target, receptor_type)
1
check_connection(this, receptor_type)
new
event
e
check_event(e)
2
connect_sender(e, receptor_type)
3
rport
rport
4
5

The connection handshake ensures valid connections

Parallelization in NEST

Model developers and users (mostly) don't have to care about parallelization.

- A neuron n is created on the virtual process p, where

$$\mathrm{gid}(n) \mod \mathrm{N}_{\mathrm{MPI}} == p$$

- On all other VPs, a light-weight proxy is created
- Devices are replicated on each VP to distribute load

- There is one random number generator (RNG) per thread
- In addition, there is a global RNG that is kept synchronized

Registering models with the kernel

- To make the new model available, add it to **NestModule**
  - Add the header and source files to **Makefile.am**
  - Add the header as include to **modelsmodule.cpp**
  - Register the new model with the simulation kernel
- This works the same for synapse and neuron models
- In principle, it is also possible to write plugins (*modules*, but the details are about to change

Model development workflow

content

Maintaining changesets

Changes to NEST can be maintained in a private git repository.

- Use **gbp-import-orig** from the *git-buildpackage* suite
- Import upstream releases into the *upstream* branch, keep your own work in the *master* branch
- Every time a new version is released, import the new sources into the *upstream* branch and merge them into *master*
- Alternatively rebase *master* on top of *upstream*

References and further reading

- The NEST Initiative homepage at www.nest-initiative.org

- Gewaltig et al. (2012) *NEST by example: An introduction to the neural simulation tool NEST*. doi:10.1007/978-94-007-3858-4_18
- Hanuschkin et al. (2010) *A general and efficient method for incorporating precise spike times in globally time-driven simulations*. doi:10.3389/fninf.2010.00113
- Kunkel et al (2012) *Meeting the memory challenges of brain-scale network simulation*. doi:10.3389/fninf.2011.00035

Please tell us about problems. We only can fix what we know of!

3D demo!