Framework for Network Co-Simulation (FNCS) Tutorial
at the 3rd Workshop on Next-Generation Analytics for the Future Power Grid

JASON FULLER, JEFF DAILY LAURENTIU MARINOVICI, ANDREW FISHER, KHUSHBU AGARWAL
Pacific Northwest National Laboratory
July 16, 2014
A good scientist is a person with original ideas.

A good engineer is a person who makes a design that works with as few original ideas as possible.

~~Freeman Dyson
What is the need?

- Smart grid brings information and communication technologies together with power systems:
  - Sensors and equipment gather information
  - Information is processed locally or centrally
  - Decisions are made based on this information

- But before deploying new technologies, it is important to understand:
  - What is the performance of a given technology?
  - How will new technologies interact with existing technologies?
  - Will assets at the distribution level negatively impact controls at the transmission level?
  - What are my communication system requirements to support an application?
  - Can applications share network bandwidth?
Traditionally, power grid and communication network domains have not resided within a single simulator with relatively equal consideration to the complexity of each.

A number of very powerful, domain-specific tools exist:
- Transmission (PSLF, Powerworld, DSATools, PST, etc.)
- Distribution (WindMil, SynerGEE, CYMDIST, OpenDSS, GridLAB-D, etc.)
- Telecommunications (OPNET, NetSim, ns-2, ns-3, OMNet++, etc.)

We do not need to recreate these tools
- Re-use existing simulators
- Libraries of models already exist
- Most are well validated
- Integrate and enjoy!!
Scalability and Co-Simulation

- Co-simulation allows for expansion of capabilities with minimal investment
  - Allows for re-use of existing software AND models
  - Enables multi-scale modeling and simulation required for understanding TC2
- FNCS is a framework for integrating simulators across multiple domains
  - Framework for Network Co-Simulation (FNCS – pronounced “Fee-nix”)
  - Developed for HPC applications across multiple platforms
Intended uses?

- **Distribution and Communications**
  - Sensor data and control (VVO, inverters, reconfiguration, etc.)
  - Demand response and retail markets

- **Transmission and Communications**
  - Wide Area Control (and Protection)
  - Phasor Measurement Unit data collection and control
  - Communication pathways and redundancy

- **Transmission, Distribution and Communications**
  - Trade-offs of distributed versus centralized controls
  - Hierarchical controls / reconfiguration during communication loss

- **Transmission, Distribution, Markets and Communications**
  - Transactive energy/ancillary markets (with distributed resources)
  - Integration of wholesale and retail markets

- **Visualization**
  - With connection to GridOPTICS
  - Generate simulated data sets for experimentation
FNCS Programming Guide

JEFF DAILY, JASON FULLER
LAURENTIU MARINOVICI, ANDREW FISHER, KHUSHBU AGARWAL
Pacific Northwest National Laboratory
July 16, 2014
FNCS Programming Guide Overview

- FNCS design goals.
- FNCS architecture overview.
  - Overview on how to integrate simulators.
  - FNCS assumptions.
- Programming with FNCS
  - Time management.
  - Object communication interface.
  - Synchronization algorithms in detail.
Challenges in power grid and communication network co-simulation

- Time synchronization.
- Differences in time scales.
- Messages between simulators should be delivered without incurring delays.
- Re-use of models.
- Integrating both transmission and distribution level simulators.

This is our goal – and we are nearly there ➔
FNCS Design Goals

- **Re-use** existing simulators as much as possible.
- Provide the environment for **rapid co-simulation development**.
- Support co-simulations for **multiple platforms**: single node, multiple nodes, clusters, cloud…
Programmers need to use the components for:
- Time management
- The communication interface

All other components are hidden to ease the programming.

FNCS is programmed in C++, and interfaces for C, Java, Fortran are provided with FNCS distribution.
Simulator core (the component that decides the next time step of the simulation) needs to be modified to use the time management component.

- **FNCS requires** control over the next time step of the simulator.
- **For simulator with large time steps (e.g., 5mins)** or for **discrete event simulators**, FNCS can modify the next time step of the simulator.
Components that will communicate with other simulators need to be modified to use the communication interface.
- Components need to be assigned unique name.
- Users need to handle the de-/serialization, or our serialization code generator can be used.
The public interfaces of FNCS:

- **Integrator** – a class that provides time management functions, framework initialization, and component registration. All methods are static, so users do not have to deal with object creations, deletion…

- **ObjectCommInterface** – provides methods for sending and receiving messages. Instances created and managed by Integrator.
Programming with FNCS - Object Hierarchy (Abstract)

Implementation differs according to the type of simulator and the algorithm

Implementation differs according to underlying comm. lib (zmq, mpi,...)

Implementation differs according to the type of simulator
Programming with FNCS - Object Hierarchy for Time Stepped Simulator

Simulator Core

Integrator

OptimisticTickSyncAlgorithm

GracePeriodCommManager

ZmqNetworkInterface

Simulator Component

ObjectCommInterface

We can switch to a different algorithm with just one function call.
Programming with FNCS - Object Hierarchy for Network Simulator

Simulator Core

Integrator

OptimisticCommSync Algorithm

Simulator Component

ObjectCommInterface

CommunicationCommManager

ZmqNetworkInterface

We can switch to a different algorithm with just one function call
Before the simulator starts a timestep, FNCS needs to be initialized.

Factory methods are called to initialize the object hierarchy according to the type of simulator and user requirements.

Properties about the simulator and co-simulation can be specified in a json file or in a function call.

```cpp
Integrator::InitIntegrator(char *jsonfile, TIME initialTime)

Integrator::InitIntegrator<syncAlgo>(timemetric simmetric, TIME initialTime, TIME packetlostperiod, TIME onetimestep,...)

Integrator::registerTimeCallback(Callback<TIME> *given)
```
FNCS requires the following from the simulator:
- Time scale,
- Initial time,
- One time step,
- A callback function that returns the current time of the simulator.

FNCS' internal time is in nanoseconds.

```c
initIntegratorGracePeriod(
    MILLISECONDS, 2300000000,
    currentTime, 10);

setregistercallback(getCurrentTime);

TIME getCurrentTime()
{
    return currentTime;
}
```

Initialize the framework for a simulator with 10 millisecond time steps.
Programming with FNCS: Initialization

- From users FNCS requires:
  - Sync algorithm to use during co-simulation.
  - Parameters of the sync algorithm.

- Init function with conservative synchronization algorithm, m:
  ```
  initIntegratorGracePeriod(
    MILLISECONDS, 2300000000, currentTime, 10);
  ```

- Init function with speculative algorithm with increasing speculation strategy and 5min initial speculation:
  ```
  initIntegratorOptimisticIncreasing(
    MILLISECONDS, 2300000000, currentTime, 300000);
  ```
Programming with FNCS: Initialization

- **Init ZMQ network interface for network simulator**
  
  ```cpp
  zmqNetworkInterface *interface=new zmqNetworkInterface(true);
  increasingSpeculationTimeStrategy *st=new increasingSpeculationTimeStrategy(NANOSECONDS,300000000000);
  Integrator::initIntegratorOptimisticCommunicate(interface,NANOSECONDS,5100000000,0,300000000000,st);
  sim_comm::CallBack<
  uint64_t,sim_comm::empty,sim_comm::empty,sim_comm::empty>
  *timerCallback=sim_comm::CreateCallback(…);
  Integrator::setTimeCallBack(timerCallback);
  ```

- **Init FNCS for a network simulator with NANOSEOND timescale and speculative sync algo.**

  - Specify a custom strategy to use for the network simulator.

- **Use increasing speculation strategy.**

  Users can specify the network interface they want to use, such as ZMQ, MPI (experimental), or extend FNCS with another network interface.

- **Register callback to the method/function that returns the current time of the simulator.**
FNCS provides two methods for time management.

- `timeStepStart()` – Called at the beginning of a time step.
- `getNextTime()` – Called at the end of a time step.

The implementation of these methods are provided in concrete syncalgorithm classes.

```
Integrator::timeStepStart(currentTime) -> wait for other simulators iff synchronization is necessary at time step currentTime.
Integrator::getNextTime(xurrentTime, nexTime) -> get the next granted time for the simulator.
```
currentTime = initialTime;
do{
    Integrator::timeStepStart(currentTime);
    processTimeStep(currentTime);
    nextTime = getNextTimeStep();
    nextTime = Integrator::getNextTime(currentTime,
     nextTime);
    currentTime = nextTime;
}while(currentTime < endTime);
while(!empty(eventQ)){
    toProcess=getNextEvent(eventQ);
    currentTime=toProcess.time;
    processEvent(toProcess);
    Integrator::timeStepStart(currentTime);
    if(!empty(eventQ))
        nextTime=getNextEvent(eventQ).time;
    else
        nextTime=Infinity;
    nextTime=Integrator::getNextTime(currentTime, nextTime);
}
Message exchange is designed to deliver messages without incurring delays (due to synchronization).

Message delivery is realized during synchronization.

- Messages are buffered until the synchronization is completed.
- The call order of the time management functions ensures messages are delivered to the network simulator on time!
At the method `getNextTime()` FNCS calculate a *Lower Bound on Time Step (LBTS)*.

LBTS is a time step until which we are sure none of the simulators are going to exchange messages.

- Calculation of LBTS is necessary for consistent delivery of inter-simulator messages.

Co-simulations can consist of simulators with different time scales.

- LBTS can be lower than the next time step of the simulator.
- Simulators with the coarser time scales will run for simulators with fine time scales.
- FNCS provides 4 algorithms for time management.
Time Management in FNCS: Conservative Algorithm

- LBTS is always set to the smallest next time step of simulators.
  1. If there are in transit messages, nextTime_i = Δt, Δt is the time-step of the ith simulator.
  2. LBTS = Reduce_min(nextTime_i)
  3. If LBTS < nextTime_i, busy wait.

- Networks simulator does not participate in calculation of LBTS.

- Networks simulator is always synchronized with the smallest next time step of the other simulators, ensuring on-time delivery.

Cons:
- Performance, at worst case needs to synchronize every time step of Sim2.
- To many message exchanges.
- Suitable for short-time co-simulations on one computer.
Time Management in FNCS: Sleeping Conservative Algorithm

- LBTS calculation is same as the conservative algorithm.

- When LBTS < nextTime_i, the simulator sleeps instead of busy waiting.
  - Simulator is woken up when LBTS == nextTime_i or when it receives a message while sleeping.
  - Reduces messages required for synchronization, which in turn increases performance.

Cons:
- Performance, at worst case needs to synchronize every time step of Sim2.
- Suitable for short-time co-simulations on multiple computers.
Time Management in FNCS: Speculative Algorithms

- A simulator can potentially send a message at \( \text{nextTime} \).
  - Conservative algorithms: Safe time synchronization choice, synchronize at \( \text{nextTime} \). (Synchronization is costly!)
  - Observation: Simulators do not need to send a message at every time step! How can we avoid synchronization at every time step without delaying message delivery?

- Speculative time synchronization algorithm:
  - Speculate that the simulators will not send messages until \( \text{specTime} >> \text{LBTS} \)
  - Fork, child processes run independently until \( \text{specTime} \), parents run the conservative algorithm.
  - Kill the children if they try to send a message before \( \text{specTime} \).
  - Kill the parents if children do not send a message until \( \text{specTime} \).
  - Fork is not costly -> uses copy on write!
  - Utilize CPU and available memory to increase the performance.
Time Management in FNCS: Speculative Algorithm

The algorithm for calculating the LBTS:

1. Calculate the number of in-transit messages (each simulator sends the number of messages sent and received).
   1. If there are in-transit messages LBTS is $\Delta t$, $\Delta t$ is the time-step of the simulator with the highest time scale.
   2. Else, LBTS is $\text{currentTime} + \Delta t_{\text{next}}$, $\Delta t_{\text{next}}$ is the minimum next time.
   3. If $\text{currentTime} + \text{LBTS} < \text{specTime}$
      1. Fork(), children use $\text{specTime}$ as LBTS
      2. If $\text{currentTime} + \text{LBTS} < \text{myNextTime}$ then enter a busy wait loop

• Cons:
  – Might not work if the simulators send messages frequently.
  – Threaded simulators need to be prepared for fork.
  – Suitable for long co-simulations where simulators do not exchange messages frequently.
Time Management in FNCS: Speculative Re-compute Algorithm

When a child process is launched, it needs to register with the FNCS broker.

- This is a costly operation!
- Reduces performance when simulators exchange messages frequently.

Speculative re-compute strategy is designed to eliminate registrations.

- Child processes are used to discover the time steps of message exchanges.

Speculation:
- \( \text{specTime} \) is always set to infinity.
- Child processes execute until one of them sends a message.
- The time of message is sent to the parent, which in turn synchronizes at this time step.

Pros:
- Does not require additional models for time synchronization.
- Improves the performance for co-simulations with simulators exchanging messages frequently.

Cons:
- Might not work if the simulators send messages frequently.
- Performance improvement not as good as speculative algorithm.
- Suitable for co-simulations where simulators exchange a lot of messages.

Key:
- Current time in the simulator.
- Next time granted by the conservative algorithm.
- Time step of message exchange.
Time Management in FNCS: Sending/Receiving Messages.

- **Register a component with FNCS**: The components of a simulator that need to communicate with other simulators need to register with FNCS.

- **Send a new message**: The component can select to be notified when it receives a message, or the messages can be stored in inbox until it reads them.

- **Get buffered messages**: Components to register a callback function.

```cpp
ObjectCommInterface *interface = Integrator::getCommInterface(<name_of_the_component>);

... Message *mesg = new Message(<from>,<to>,<timestep>,<data>,<direct_or_network>);
Interface->send(mesg);

... while(interface->hasMoreMessage()){
    Message *rm = interface->getNextMessage();
    //process rm.
}
```
Programming with FNCS

- **Extension points of FNCS:**
  - **BufferStrategy:** defines how messages are buffered.
  - **SynchronizationAlgorithm:** FNCS provides 4 synchronization algorithms, but users can extend the framework with an algorithm suitable for their needs.
  - **SpeculationStrategy:** FNCS provides synchronization based-on speculative execution to speed up co-simulations. Users can extend FNCS with strategies describing when to speculate.
  - **NetworkInterface:** FNCS provides a well-defined interface for co-simulation inter-process communication. Users can extend interface to utilize an inter-process communication library. Currently, ZMQ is supported, experimental support for MPI.
Demand Response/Real-Time Pricing Example

JASON FULLER, JEFF DAILY
LAURENTIU MARINOVICI, ANDREW FISHER, KHUSHBU AGARWAL
Pacific Northwest National Laboratory
July 16, 2014
Demand Response/Real-Time Pricing Example

- RTP, double-auction, retail market
  - Market accepts demand and supply bids
  - Clears on five minute intervals
  - Designed to also manage capacity constraints at substation

- Residential energy management system
  - Acts as a distributed agent to offer bids & respond to clearing prices
  - Consumer sets a preference for “savings” versus “comfort”

- Currently being tested as part of the AEP gridSMART® ARRA Demonstration in Columbus, OH
Basic Real-Time Price / Double Auction Market – *Typical Unconstrained Conditions*

- Market clears every 5-mins to ~match AC load cycle
- Cleared load varies with demand curve
- Clearing price is constant at base retail price

**Unresponsive Loads**
- Base retail price based on PJM 5-min real-time market
- Varies every 5-min

**Responsive Loads**
- Demand Curve: sorted (P, Q) bids from RTP-DA customers
- Feeder Supply Curve

**Feeder Capacity**
- Q_{clear}
- P_{clear}
Basic Real-Time Price / Double Auction Market – **Typical Constrained Conditions**

- **Unresponsive Loads**
  - Base retail price based on PJM 5-min real-time market
  - Feeder Supply Curve

- **Responsive Loads**
  - Demand Curve: sorted \((P, Q)\) bids from RTP-DA customers
  - Clearing price varies to keep load at capacity

- Market clears every 5-mins
- Cleared load is constant at feeder capacity

- Feeder Capacity

- \(P_{\text{clear}}\)
- \(P_{\text{base}}\)
- \(Q_{\text{clear}}\)

\(P, \text{Price} ($/MWh)\)

\(Q, \text{Load (MW)}\)
- Decreased wholesale energy costs
- Peak demand limited to feeder capacity
But what happens when including communication latency?

- IEEE-13 node model with 900 residential loads and controllers modeled in GridLAB-D
- Model was modified to work within FNCS framework
- An ns-3 communication network model was created (radial WIFI)
- EXTREME communication delays (for Wifi) were considered
But what happens when including communication latency?

- Excessive communication delays during critical period caused an “accounting error” in auction (this was considered in Demo deployment)

As simulated in GridLAB-D and ns-3
A few comments

- These communication concerns were dealt with during the design of the demonstration system
  - However, it was mostly engineering judgment and the timescale of control is such that latency is not a major factor

- A co-simulation environment can help determine the most economic means of deploying smart grid technologies, specifically in terms of communication requirements for successful system operations
  - How much communication infrastructure do I need?
  - What affect will latency have on my monitoring / control scheme?

- This will become more important as
  - Sampling / control action periods are decreased (real-time control)
  - Multiple applications are layered over the same communication systems
Now let’s add a transmission element…

- Want to be able to integrate >2 simulators
  - ns-3™
  - GridLAB-D™
  - transmission solver

- Example: Wide Area Monitoring, Protection, and Control (WAMPAC)

- Want to limit the power flowing through branch $3_4$

- Use a price “signal” broadcasted to a distribution circuit to limit demand
What data is being exchanged?

- GridLAB-D is posting current load to a transmission substation
- The transmission solver is performing power flow calculations with updated load information
- The control object is calculating the change in price needed
- A new price is being broadcasted to distributed devices in GridLAB-D via ns-3
Relatively simple control design

Simple PI control design
- Only used to show how the software works (does not deal with revenue, “price as a signal”, regulatory issues, etc.)

Demand response consumers are using the same mechanism as previous use case

“Price” is now derived as a function of the system constraints
Some results

- PI controller takes some time to learn the necessary price adjustment (not well tuned)
  - In actual application, we would take some time to tune the parameters

- But, we can see the response within GridLAB-D
  - Reduces the demand from hour 40 through 46
  - Price signal is being produced in the transmission solver (this could be replaced with Matpower and LMPs)
  - Price is broadcasted via ns-3 (we could look at affects of communication delays)
Closing Remarks

JEFF DAILY, JASON FULLER, LAURENTIU MARINOVICI, ANDREW FISHER, KHUSHBU AGARWAL
Pacific Northwest National Laboratory
July 16, 2014
Closing remarks

- Simple example(s) to demonstrate the simulation environment
  - Any tool could be replaced with another of “better value”
  - Complexity of design is up to user

- We will continue developing interconnections for further experimentation and additional use cases
  - Exploring interface with GridPACK solvers
  - Expanding MATPOWER connection
  - Expanding GridLAB-D connection
  - Finishing the interface for EnergyPlus
  - Adding an interface to GridOPTICS (for data management and visualization)

- Suggestions?
Sources soon to be on GridOPTICS github site.

- [https://github.com/GridOPTICS/FNCS](https://github.com/GridOPTICS/FNCS) (empty placeholder)
- Can use issue tracker right now
- Code rollout is underway

Email us developers directly

- [jeff.daily@pnnl.gov](mailto:jeff.daily@pnnl.gov), PI
- [jason.fuller@pnnl.gov](mailto:jason.fuller@pnnl.gov), co-PI