Introduction to Psyllid

Noah S. Oblath

February 1, 2021

What is Psyllid?

- DAQ software package for Project 8
- Contains various modular tools adapted for Project 8 data
- Framework is also used by ADMX
- Written in C++
- User interface relies on the Dripline communication standard

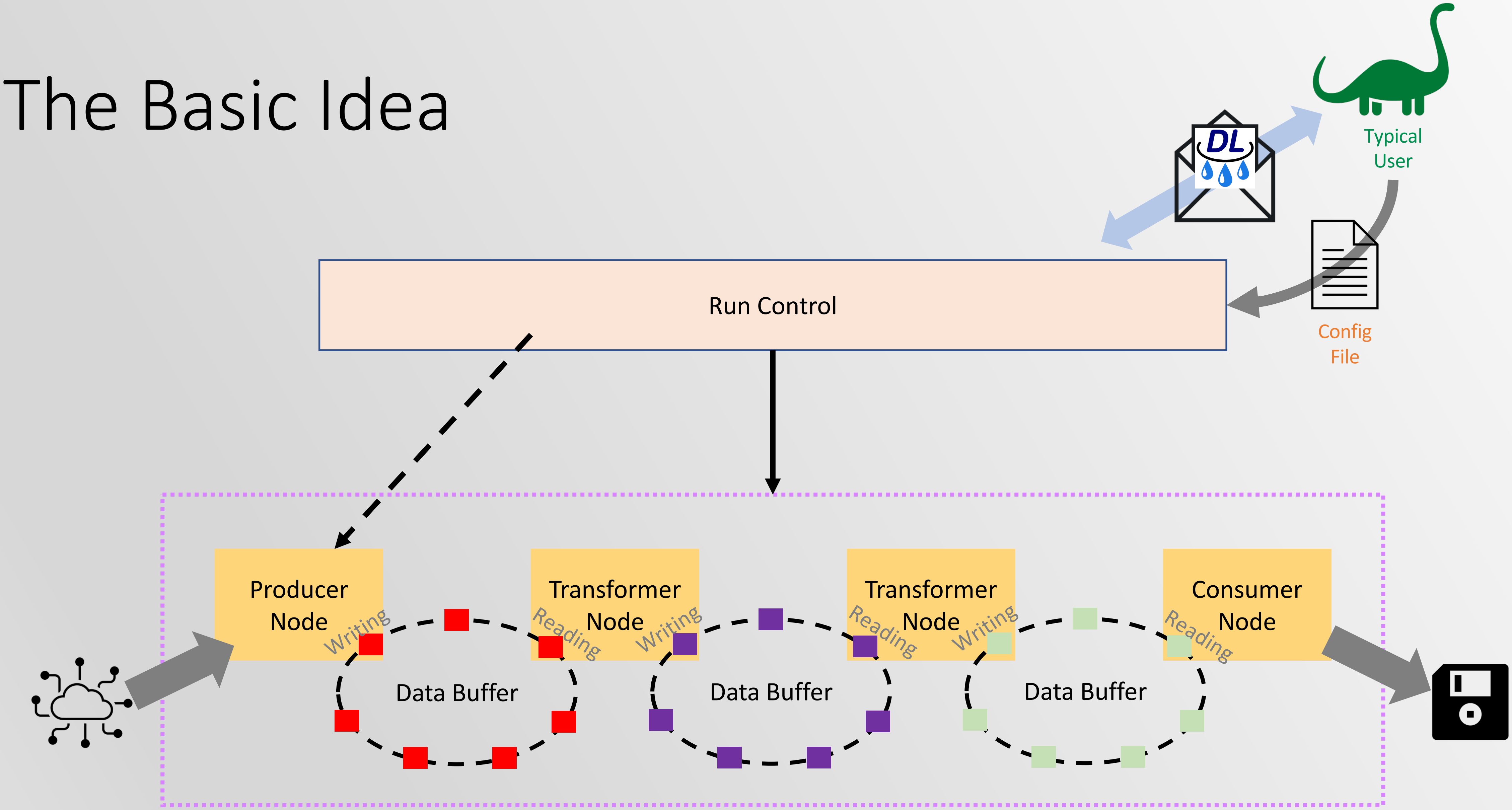
Resources

- Git repository: https://github.project8.org/psyllid
- Documentation: https://project8.org/psyllid
- Code documentation: https://psyllid.readthedocs.io/en/master/_static/index.html
- Dripline documentation: https://driplineorg.github.io
- Slack channel: #psyllid

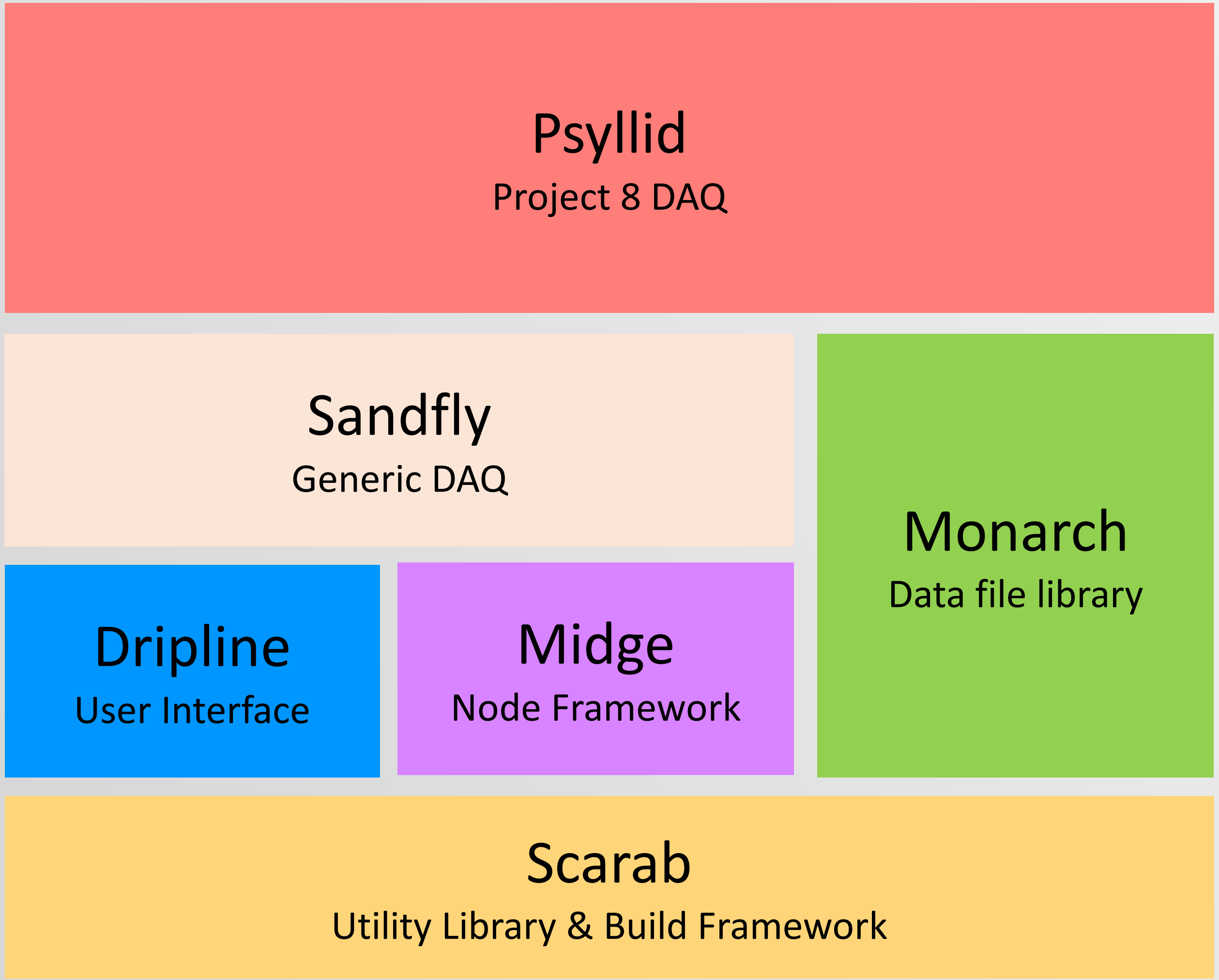
Objectives

- To understand . . .
- The basic framework used
- How to run Psyllid
- Understand the contents and operation of a node

The Basic Idea



Software Stack



Psyllid

Project 8 DAQ

Sandfly

Generic DAQ

Monarch

Data file library

Dripline

User Interface

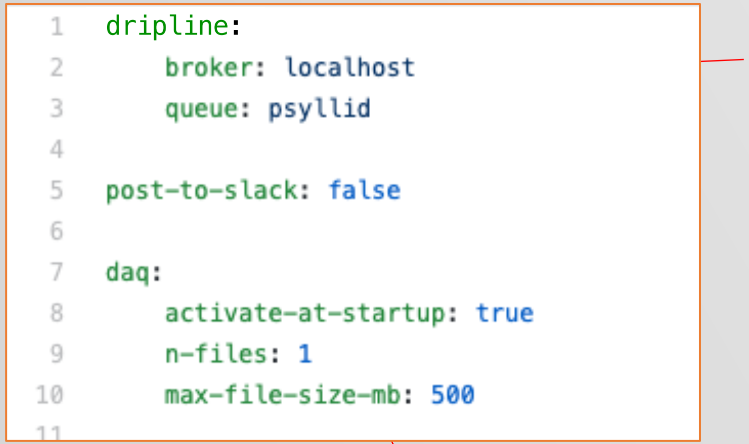
Midge

Node Framework

Scarab

Utility Library & Build Framework

Example Config File



```
dripline:
    broker: localhost
    queue: psyllid

post-to-slack: false

daq:
    activate-at-startup: true
    n-files: 1
    max-file-size-mb: 500
```

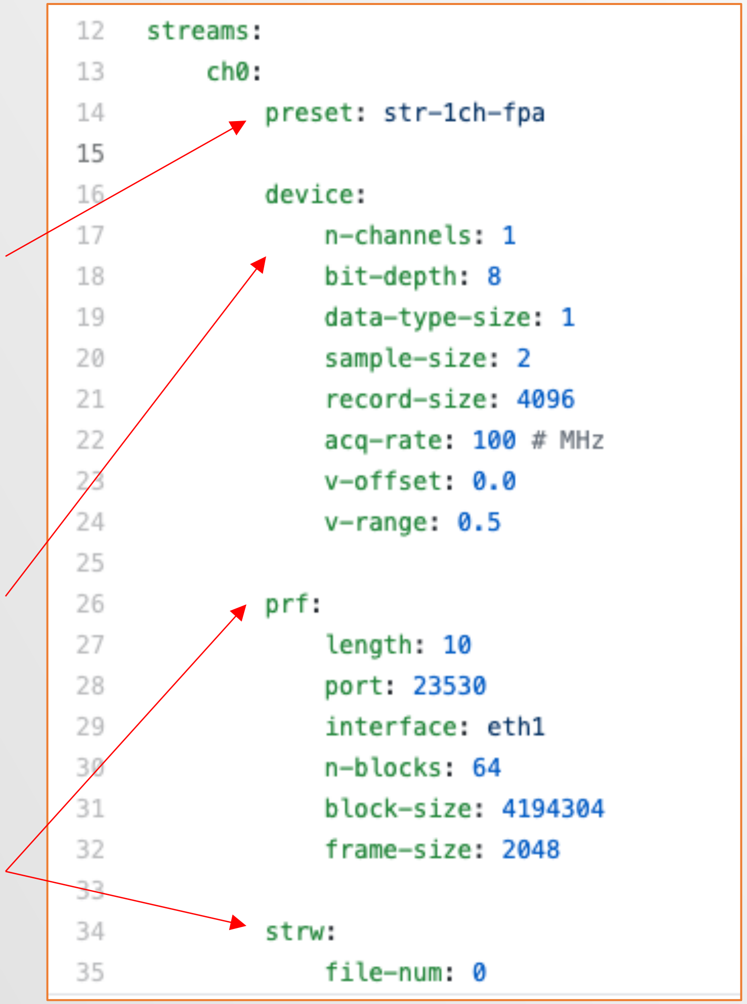
Dripline interface details

Global settings

Sets up nodes:
`packet_receiver_fpa`
`tf_roach_receiver`
`streaming_writer`
`term_freq_data`

Information about data source

Node configurations



```
streams:
    ch0:
        preset: str-1ch-fpa

        device:
            n-channels: 1
            bit-depth: 8
            data-type-size: 1
            sample-size: 2
            record-size: 4096
            acq-rate: 100 # MHz
            v-offset: 0.0
            v-range: 0.5

        prf:
            length: 10
            port: 23530
            interface: eth1
            n-blocks: 64
            block-size: 4194304
            frame-size: 2048

        strw:
            file-num: 0
```

What Nodes are in Psyllid Already?

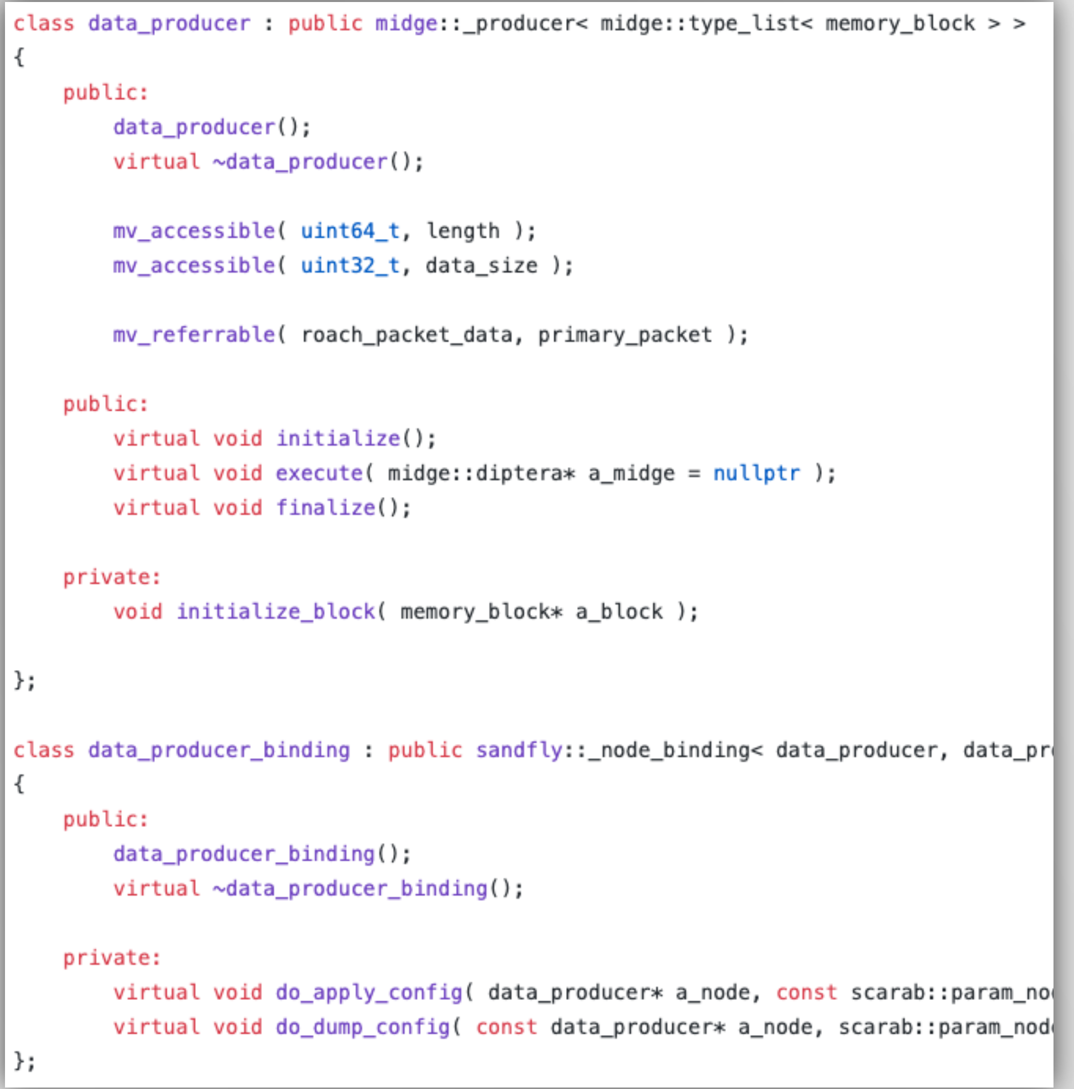
- data_producer: blank data generator
- egg3_reader: reads egg3 files (e.g. simulated)
- event_builder: determines which records go together as an event
- frequency_mask_trigger: marks triggered records
- frequency_transform: FFT
- packet_receiver_fpa: fast-packet acquisition
- packet_receiver_socket: standard network interface
- streaming_frequency_writer: writes frequency data to an egg3 file (non-standard use)
- streaming_writer: writes time-domain data to an egg3 file
- terminator: stops a chain
- tf_roach_receiver: interprets Phase II ROACH data and splits time and frequency paths
- triggered_writer: writes triggered data to an egg3 file

Priorities: Psyllid vs Katydid

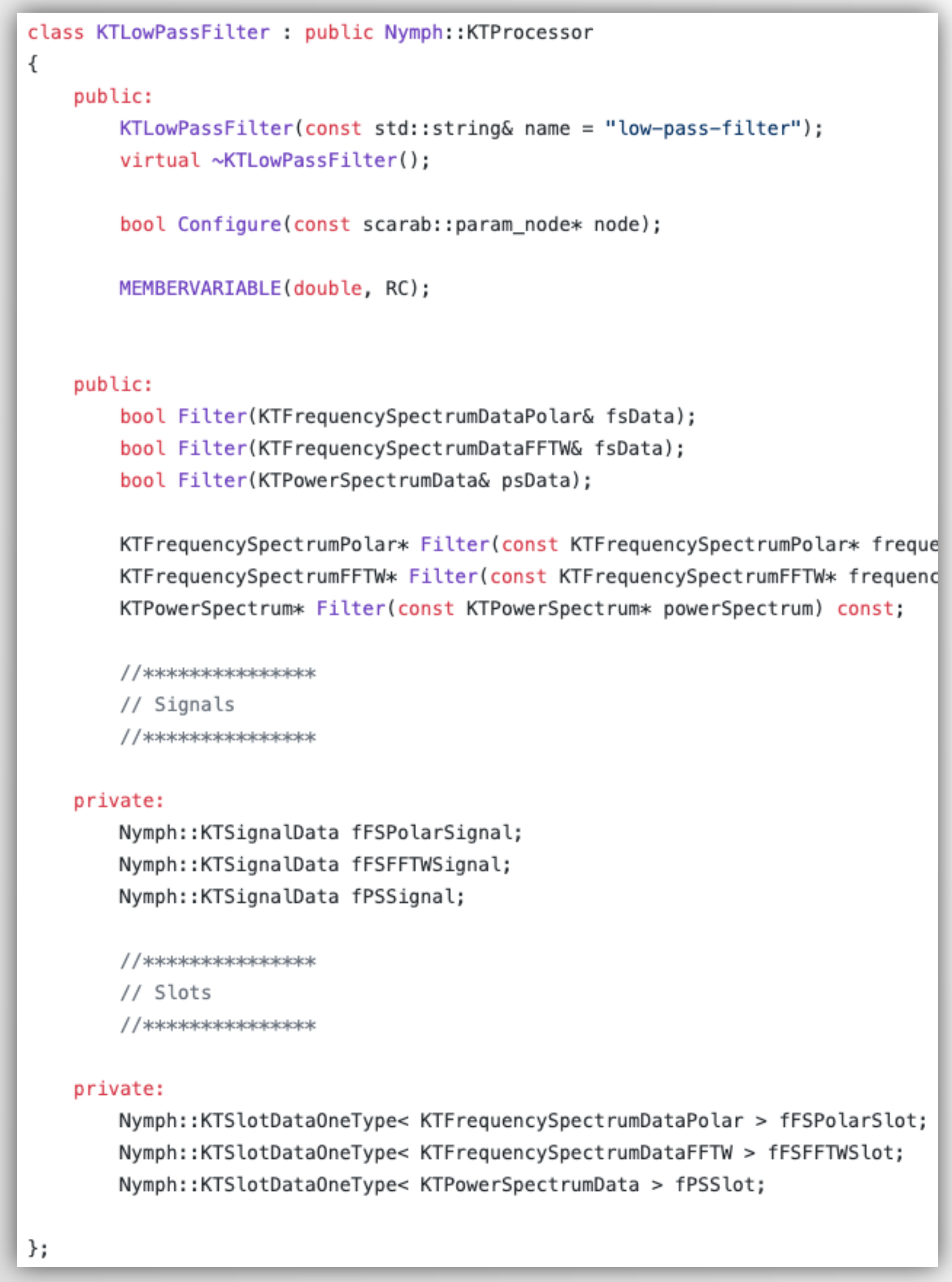
	Psyllid	Katydid
Speed	High	Medium
Flexibility	Medium	High
Modularity	High	High
Operation	Continuous	Discrete

Psyllid vs. Katydid:

Node vs. Processor

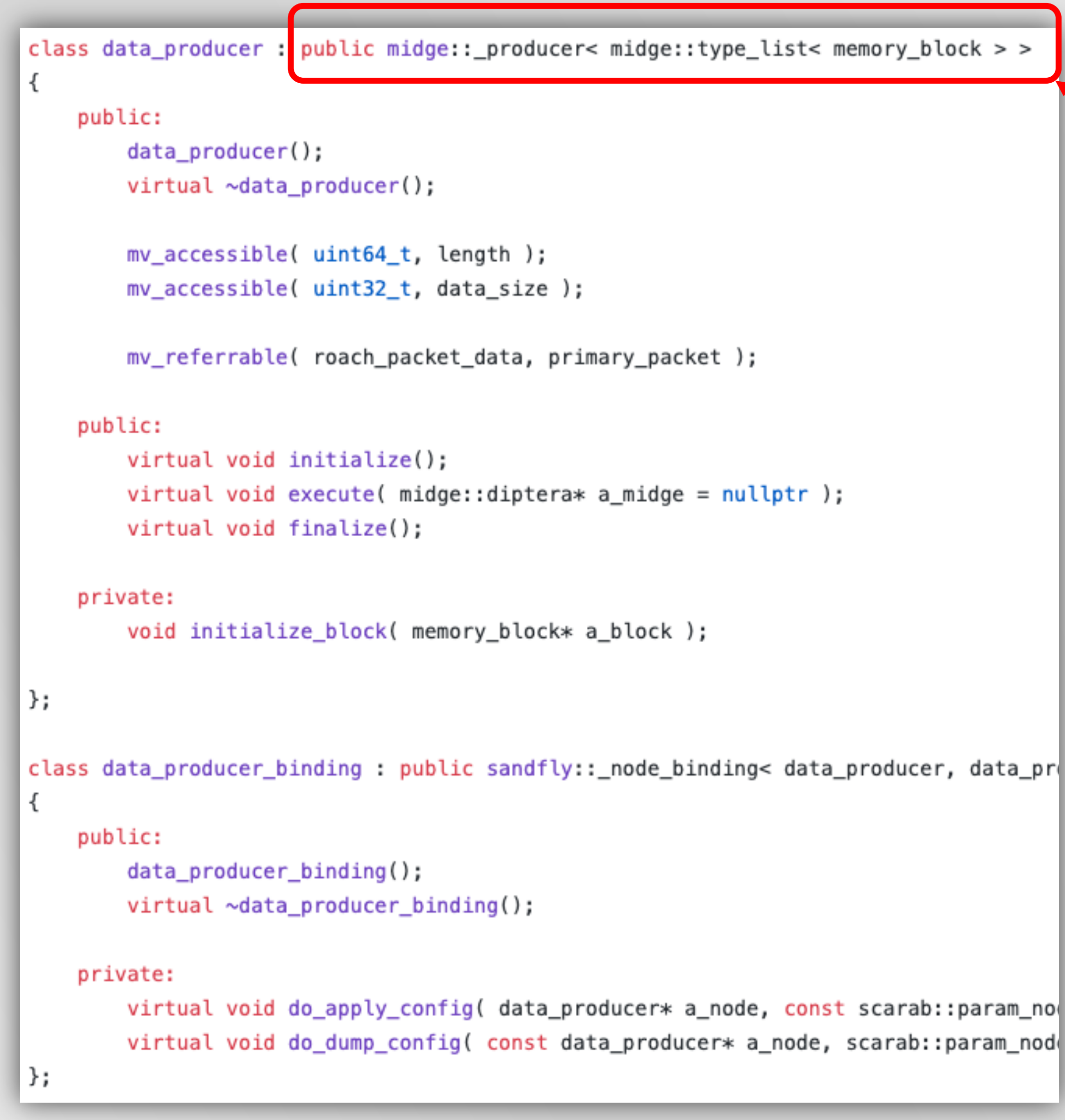


```
class data_producer : public midge::_producer< midge::type_list< memory_block > >
{
    public:
        data_producer();
        virtual ~data_producer();

        mv_accessible( uint64_t, length );
        mv_accessible( uint32_t, data_size );

        mv_referrable( roach_packet_data, primary_packet );

    public:
        virtual void initialize();
        virtual void execute( midge::diptera* a_midge = nullptr );
        virtual void finalize();

    private:
        void initialize_block( memory_block* a_block );

};

class data_producer_binding : public sandfly::_node_binding< data_producer, data_pr
{
    public:
        data_producer_binding();
        virtual ~data_producer_binding();

    private:
        virtual void do_apply_config( data_producer* a_node, const scarab::param_no
        virtual void do_dump_config( const data_producer* a_node, scarab::param_nod
};
```



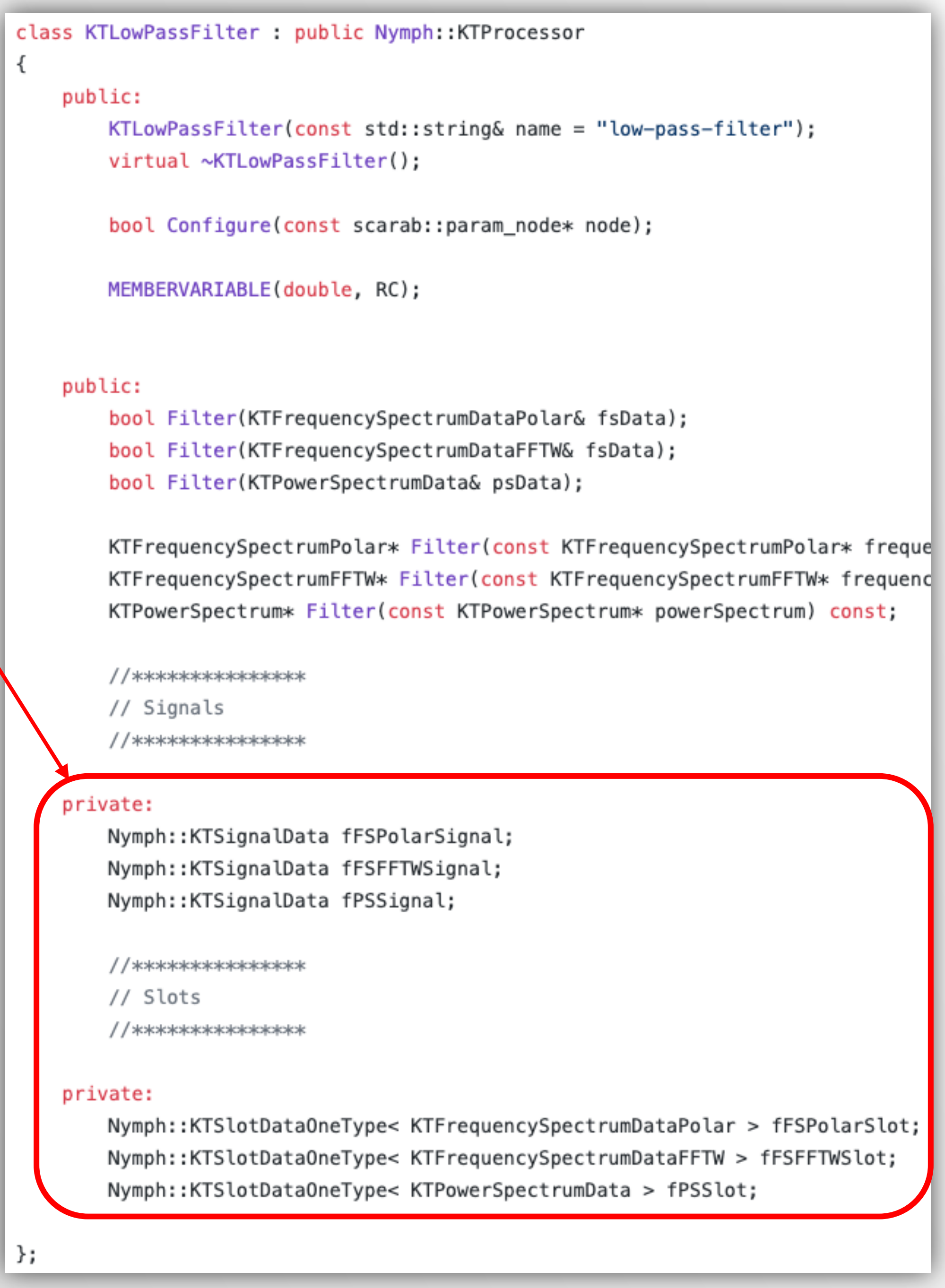
```
class KTLowPassFilter : public Nymph::KTProcessor
{
    public:
        KTLowPassFilter(const std::string& name = "low-pass-filter");
        virtual ~KTLowPassFilter();

        bool Configure(const scarab::param_node* node);

        MEMBERVARIABLE(double, RC);


    public:
        bool Filter(KTFrequencySpectrumDataPolar& fsData);
        bool Filter(KTFrequencySpectrumDataFFTW& fsData);
        bool Filter(KTPowerSpectrumData& psData);

        KTFrequencySpectrumPolar* Filter(const KTFrequencySpectrumPolar* freque
        KTFrequencySpectrumFFTW* Filter(const KTFrequencySpectrumFFTW* frequenc
        KTPowerSpectrum* Filter(const KTPowerSpectrum* powerSpectrum) const;

        //***************
        // Signals
        //***************

    private:
        Nymph::KTSignalData fFSPolarSignal;
        Nymph::KTSignalData fFSFFTWSignal;
        Nymph::KTSignalData fPSSignal;

        //***************
        // Slots
        //***************

    private:
        Nymph::KTSlotDataOneType< KTFrequencySpectrumDataPolar > fFSPolarSlot;
        Nymph::KTSlotDataOneType< KTFrequencySpectrumDataFFTW > fFSFFTWSlot;
        Nymph::KTSlotDataOneType< KTPowerSpectrumData > fPSSlot;

};
```

Psyllid vs. Katydid:
Node vs. Processor



```
class data_producer : public midge::_producer< midge::type_list< memory_block > >
{
    public:
        data_producer();
        virtual ~data_producer();

        mv_accessible( uint64_t, length );
        mv_accessible( uint32_t, data_size );

        mv_referrable( roach_packet_data, primary_packet );

    public:
        virtual void initialize();
        virtual void execute( midge::diptera* a_midge = nullptr );
        virtual void finalize();

    private:
        void initialize_block( memory_block* a_block );

};

class data_producer_binding : public sandfly::_node_binding< data_producer, data_pr
{
    public:
        data_producer_binding();
        virtual ~data_producer_binding();

    private:
        virtual void do_apply_config( data_producer* a_node, const scarab::param_no
        virtual void do_dump_config( const data_producer* a_node, scarab::param_nod
};
```

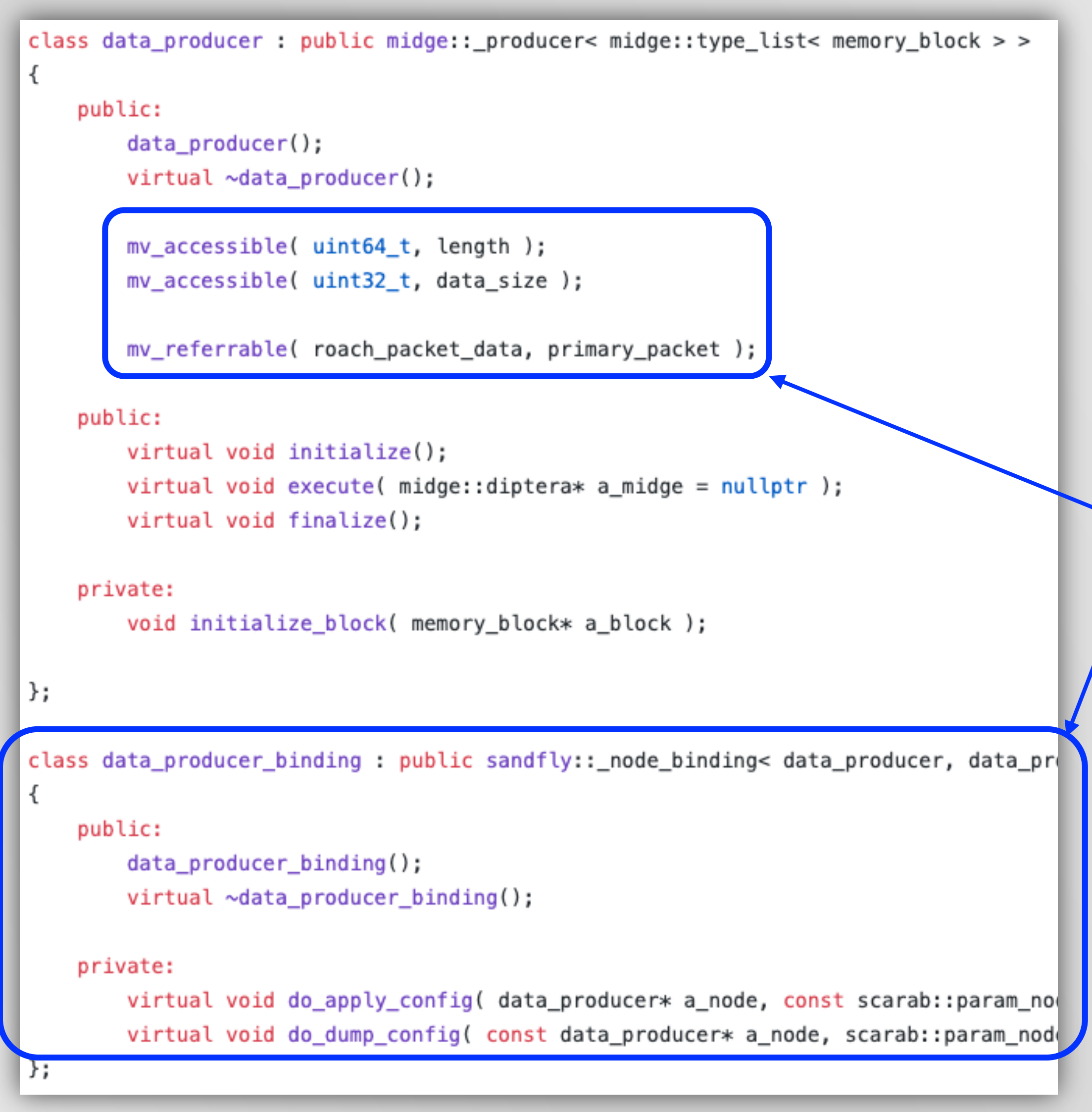
Framework integration



```
class KTLowPassFilter : public Nymph::KTProcessor
{
    public:
        KTLowPassFilter(const std::string& name = "low-pass-filter");
        virtual ~KTLowPassFilter();

        bool Configure(const scarab::param_node* node);

        MEMBERVARIABLE(double, RC);


    public:
        bool Filter(KTFrequencySpectrumDataPolar& fsData);
        bool Filter(KTFrequencySpectrumDataFFTW& fsData);
        bool Filter(KTPowerSpectrumData& psData);

        KTFrequencySpectrumPolar* Filter(const KTFrequencySpectrumPolar* freque
        KTFrequencySpectrumFFTW* Filter(const KTFrequencySpectrumFFTW* frequenc
        KTPowerSpectrum* Filter(const KTPowerSpectrum* powerSpectrum) const;

        //***************
        // Signals
        //***************

    private:
        Nymph::KTSignalData fFSPolarSignal;
        Nymph::KTSignalData fFSFFTWSignal;
        Nymph::KTSignalData fPSSignal;

        //***************
        // Slots
        //***************

    private:
        Nymph::KTSlotDataOneType< KTFrequencySpectrumDataPolar > fFSPolarSlot;
        Nymph::KTSlotDataOneType< KTFrequencySpectrumDataFFTW > fFSFFTWSlot;
        Nymph::KTSlotDataOneType< KTPowerSpectrumData > fPSSlot;

};
```

Psyllid vs. Katydid:

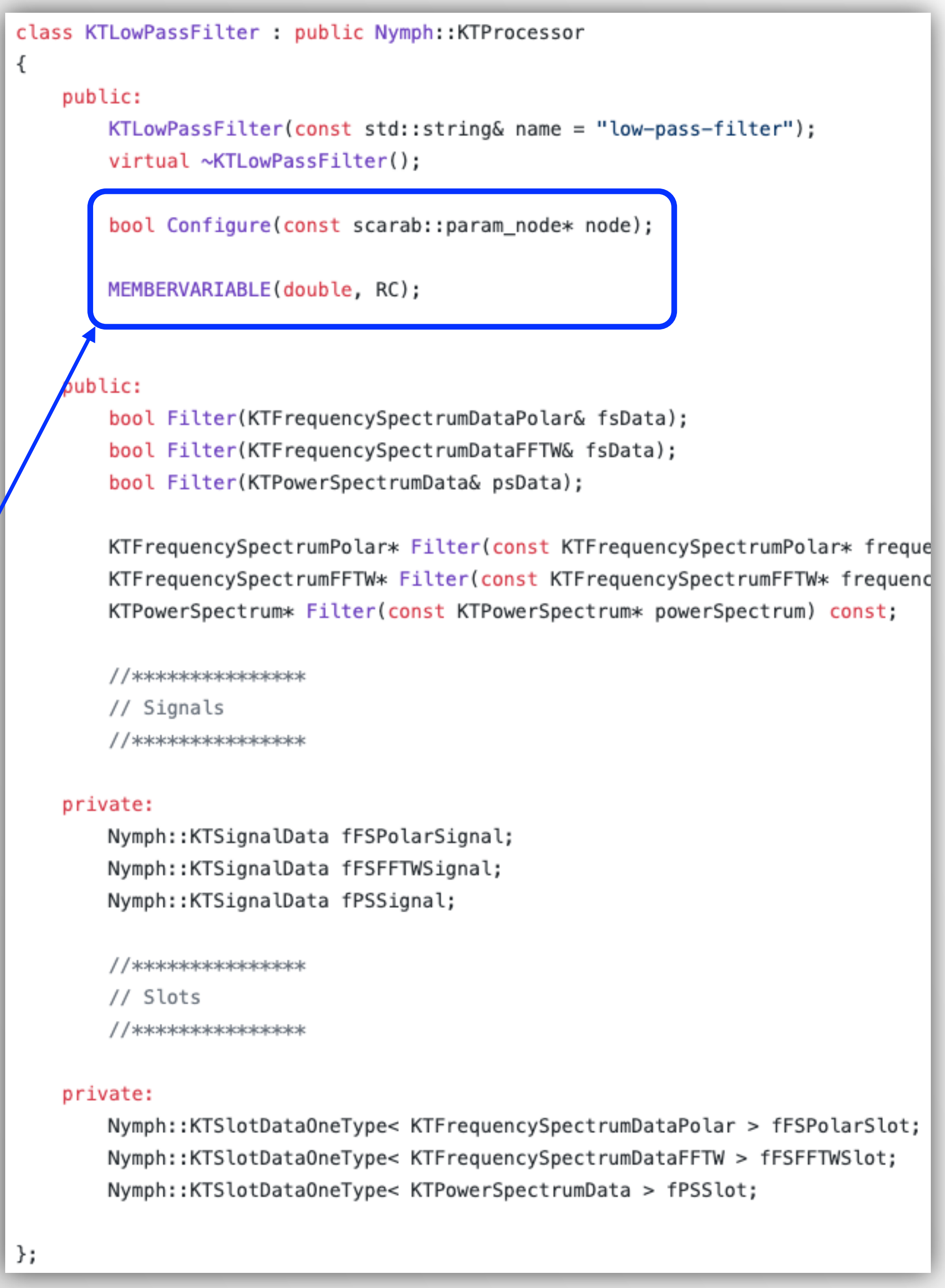
Node vs. Processor



```
class data_producer : public midge::_producer< midge::type_list< memory_block > >
{
    public:
        data_producer();
        virtual ~data_producer();

        mv_accessible( uint64_t, length );
        mv_accessible( uint32_t, data_size );

        mv_referrable( roach_packet_data, primary_packet );

    public:
        virtual void initialize();
        virtual void execute( midge::diptera* a_midge = nullptr );
        virtual void finalize();

    private:
        void initialize_block( memory_block* a_block );

};
```

```
class data_producer_binding : public sandfly::_node_binding< data_producer, data_pr
{
    public:
        data_producer_binding();
        virtual ~data_producer_binding();

    private:
        virtual void do_apply_config( data_producer* a_node, const scarab::param_no
        virtual void do_dump_config( const data_producer* a_node, scarab::param_nod
};
```

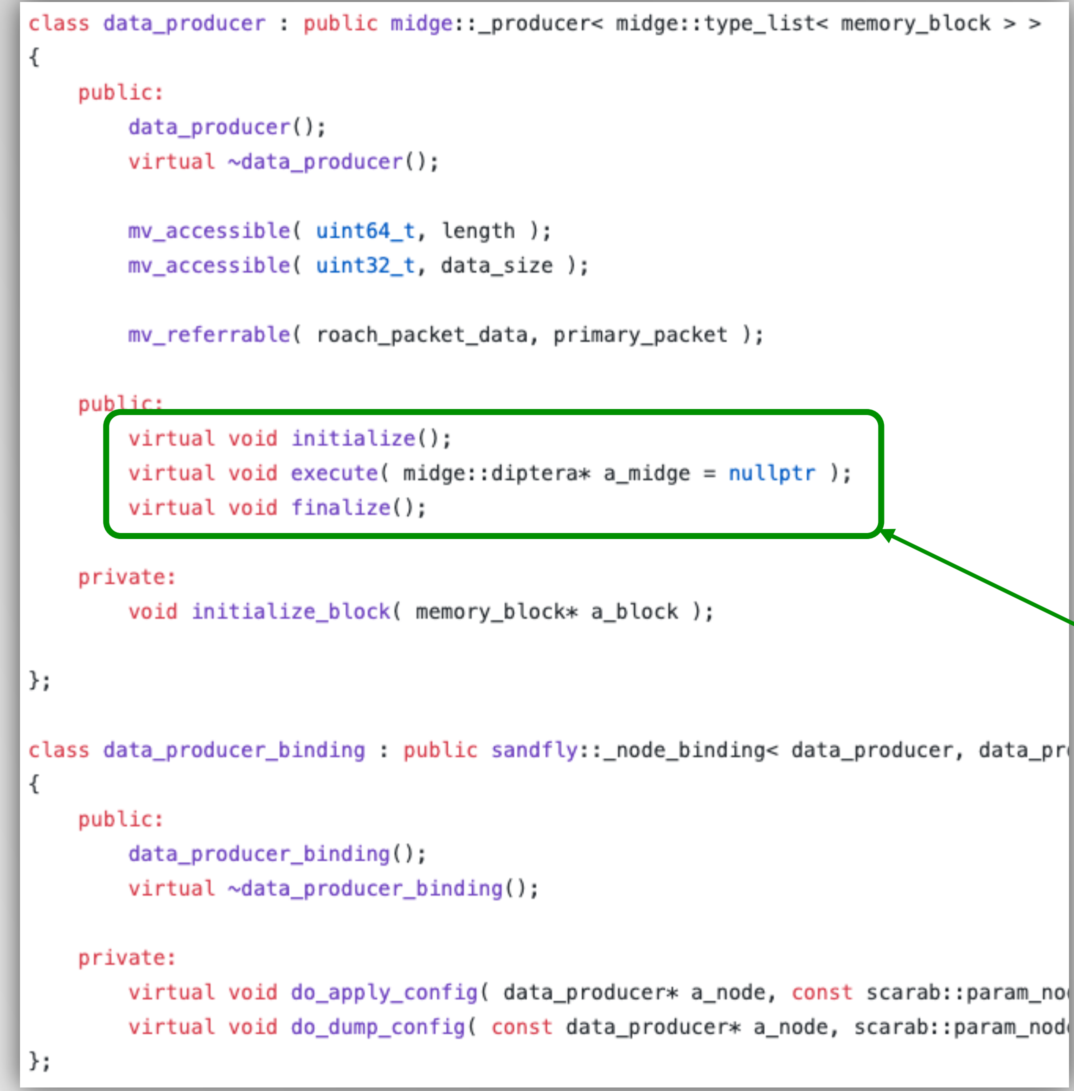
Configuration



```
class KTLowPassFilter : public Nymph::KTProcessor
{
    public:
        KTLowPassFilter(const std::string& name = "low-pass-filter");
        virtual ~KTLowPassFilter();

        bool Configure(const scarab::param_node* node);

        MEMBERVARIABLE(double, RC);

    public:
        bool Filter(KTFrequencySpectrumDataPolar& fsData);
        bool Filter(KTFrequencySpectrumDataFFTW& fsData);
        bool Filter(KTPowerSpectrumData& psData);

        KTFrequencySpectrumPolar* Filter(const KTFrequencySpectrumPolar* freque
        KTFrequencySpectrumFFTW* Filter(const KTFrequencySpectrumFFTW* frequenc
        KTPowerSpectrum* Filter(const KTPowerSpectrum* powerSpectrum) const;

        //***************
        // Signals
        //***************

    private:
        Nymph::KTSignalData fFSPolarSignal;
        Nymph::KTSignalData fFSFFTWSignal;
        Nymph::KTSignalData fPSSignal;

        //***************
        // Slots
        //***************

    private:
        Nymph::KTSlotDataOneType< KTFrequencySpectrumDataPolar > fFSPolarSlot;
        Nymph::KTSlotDataOneType< KTFrequencySpectrumDataFFTW > fFSFFTWSlot;
        Nymph::KTSlotDataOneType< KTPowerSpectrumData > fPSSlot;

};
```

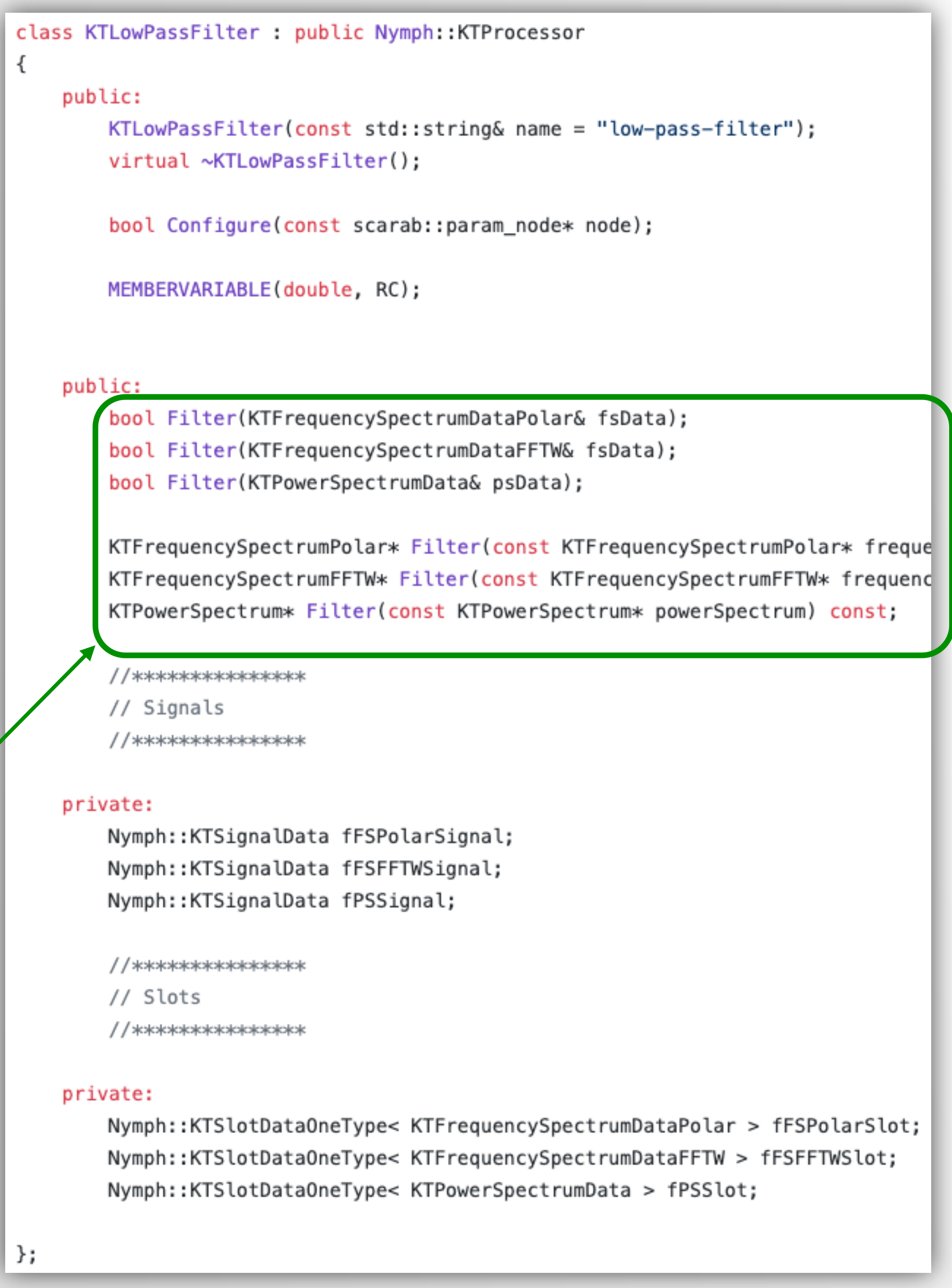
Psyllid vs. Katydid:

Node vs. Processor



```
class data_producer : public midge::_producer< midge::type_list< memory_block > >
{
    public:
        data_producer();
        virtual ~data_producer();

        mv_accessible( uint64_t, length );
        mv_accessible( uint32_t, data_size );

        mv_referrable( roach_packet_data, primary_packet );

    public:
        virtual void initialize();
        virtual void execute( midge::diptera* a_midge = nullptr );
        virtual void finalize();

    private:
        void initialize_block( memory_block* a_block );

};

class data_producer_binding : public sandfly::_node_binding< data_producer, data_pr
{
    public:
        data_producer_binding();
        virtual ~data_producer_binding();

    private:
        virtual void do_apply_config( data_producer* a_node, const scarab::param_no
        virtual void do_dump_config( const data_producer* a_node, scarab::param_nod
};
```

Action

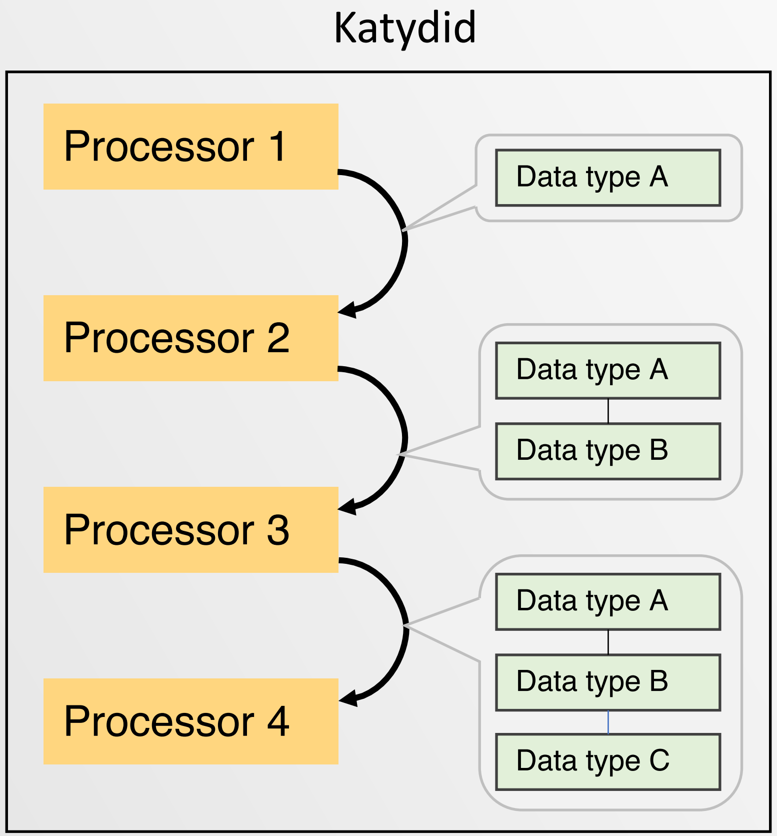


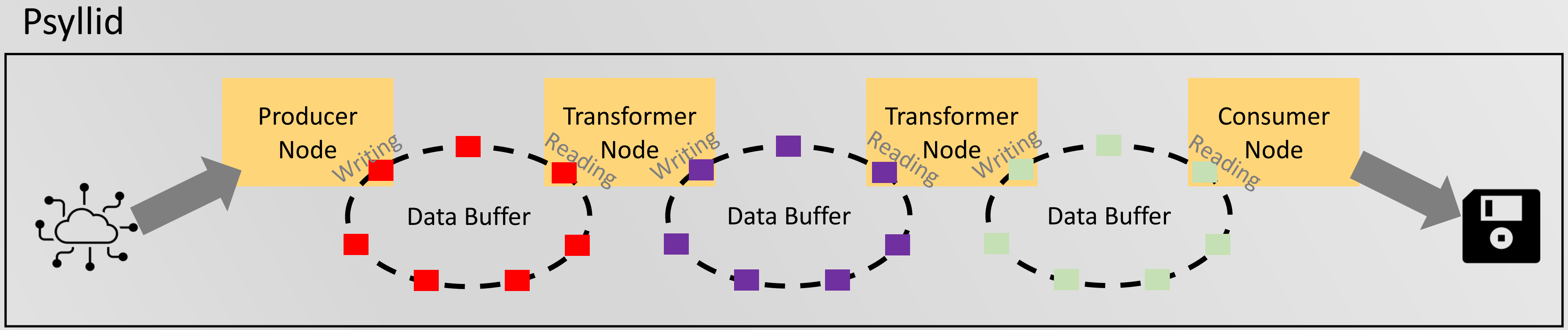
```
class KTLowPassFilter : public Nymph::KTProcessor
{
    public:
        KTLowPassFilter(const std::string& name = "low-pass-filter");
        virtual ~KTLowPassFilter();

        bool Configure(const scarab::param_node* node);

        MEMBERVARIABLE(double, RC);


    public:
        bool Filter(KTFrequencySpectrumDataPolar& fsData);
        bool Filter(KTFrequencySpectrumDataFFTW& fsData);
        bool Filter(KTPowerSpectrumData& psData);

        KTFrequencySpectrumPolar* Filter(const KTFrequencySpectrumPolar* freque
        KTFrequencySpectrumFFTW* Filter(const KTFrequencySpectrumFFTW* frequenc
        KTPowerSpectrum* Filter(const KTPowerSpectrum* powerSpectrum) const;

        //***************
        // Signals
        //***************

    private:
        Nymph::KTSignalData fFSPolarSignal;
        Nymph::KTSignalData fFSFFTWSignal;
        Nymph::KTSignalData fPSSignal;

        //***************
        // Slots
        //***************

    private:
        Nymph::KTSlotDataOneType< KTFrequencySpectrumDataPolar > fFSPolarSlot;
        Nymph::KTSlotDataOneType< KTFrequencySpectrumDataFFTW > fFSFFTWSlot;
        Nymph::KTSlotDataOneType< KTPowerSpectrumData > fPSSlot;

};
```

Psyllid vs. Katydid: Data Flow

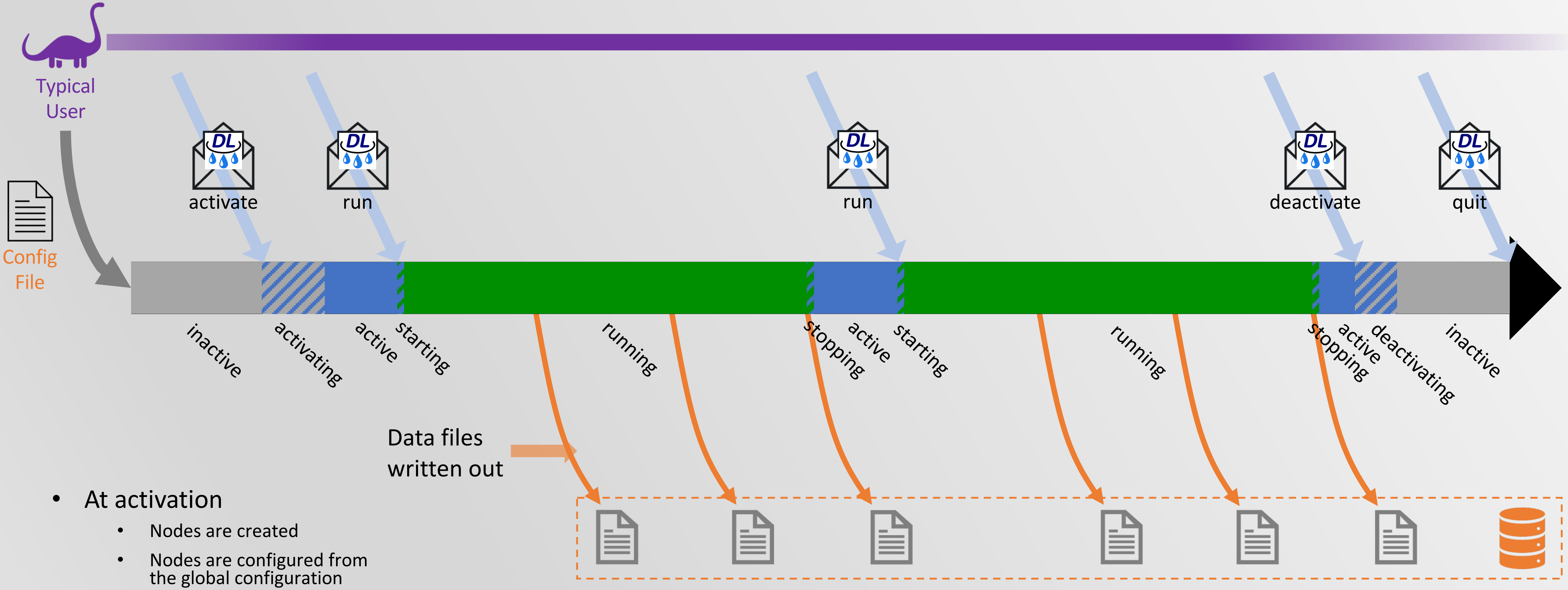
- **Katydid**: there's only one* event loop
- **Psyllid**: each node is an event loop
- **Katydid**: data are processed in sequential function calls
- **Psyllid**: data are processed independently in separate threads

* Katydid can be configured with multiple event loops, but typically only for major data paradigm changes





When You Run Psyllid



- At activation
  - Nodes are created
  - Nodes are configured from the global configuration
  - Nodes are started and paused
- At run start
  - Nodes are unpaused