Algorithms for
Smart broadcasting

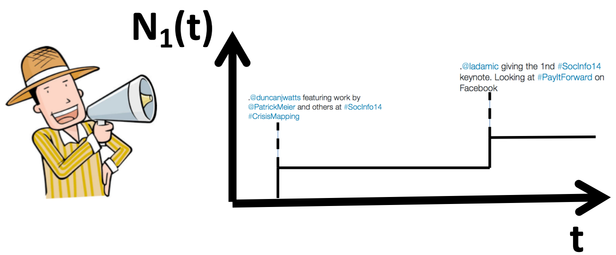
HUMAN-CENTERED MACHINE LEARNING

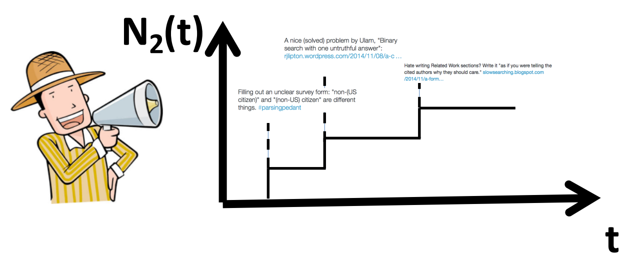
http://courses.mpi-sws.org/hcml-ws18/

MAX PLANCK INSTITUTE
FOR SOFTWARE SYSTEMS

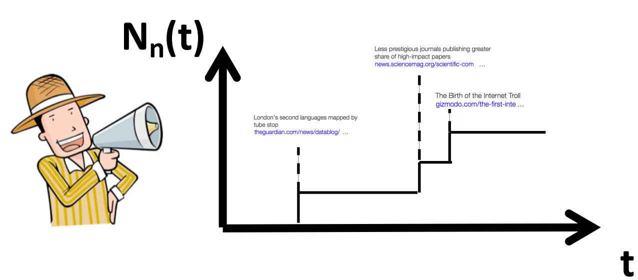
Recap: *When-to-post* problem setup

Broadcasters' posts as a counting process N(t)



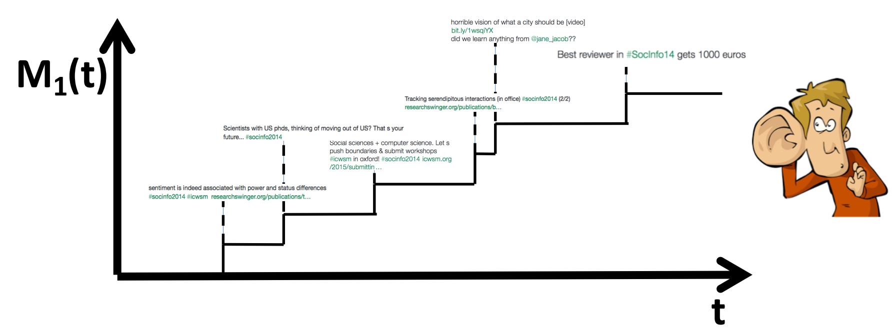


⋮

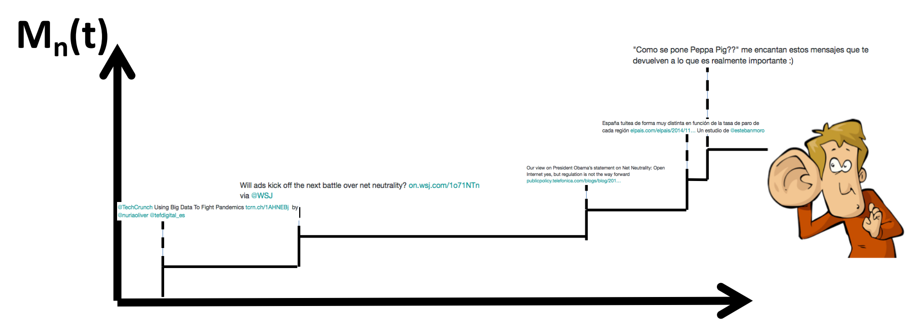


$$M(t) = A^T N(t)$$

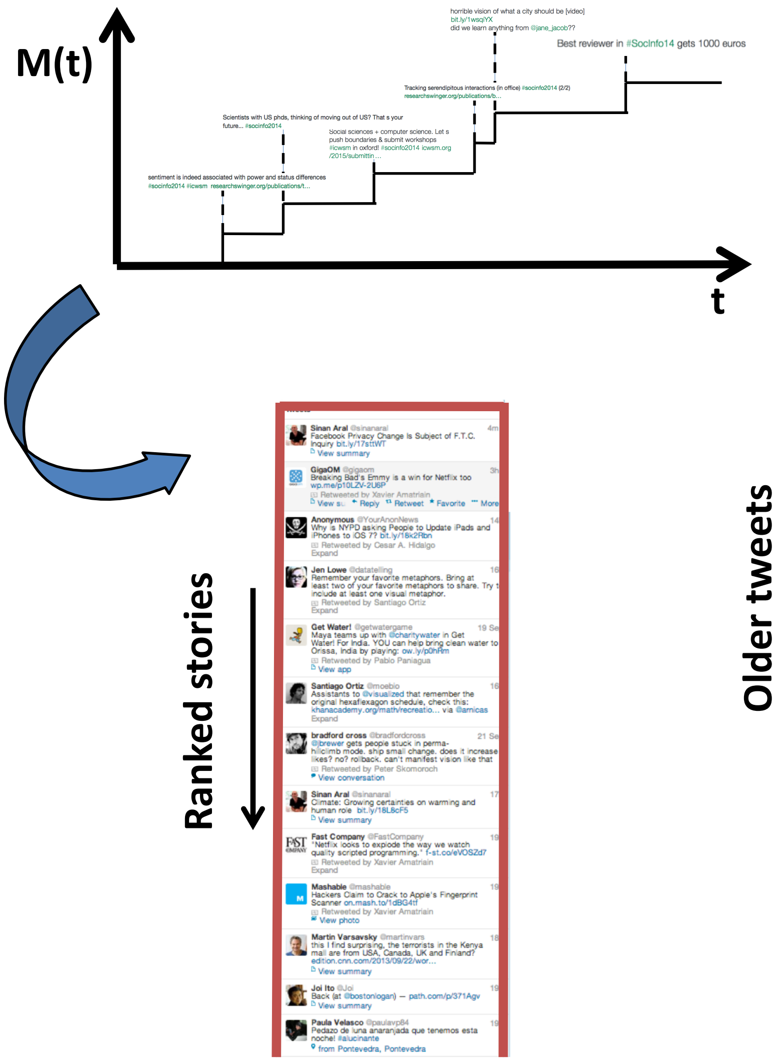
Users' feeds as sum of counting processes M(t)



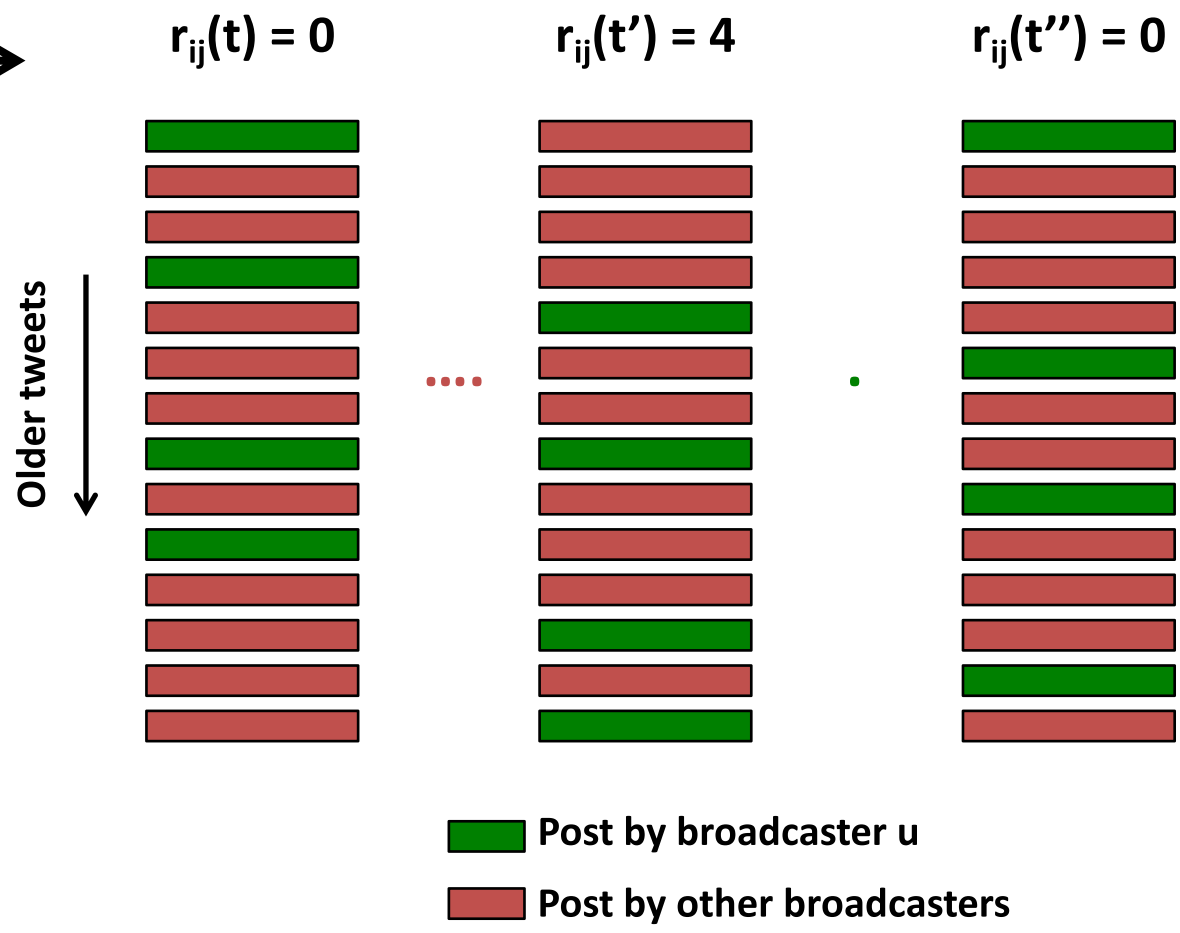
⋮



Recap: Measuring Visibility



Position of the highest ranked tweet by broadcaster i in follower j's wall



Recap: Maximizing Visibility

Minimize (quadratic) loss:

Posting rate

$$\text{minimize} \int_0^T \left(r^2(t) + c\,\lambda^2(t)\right)\, dt$$

Rank

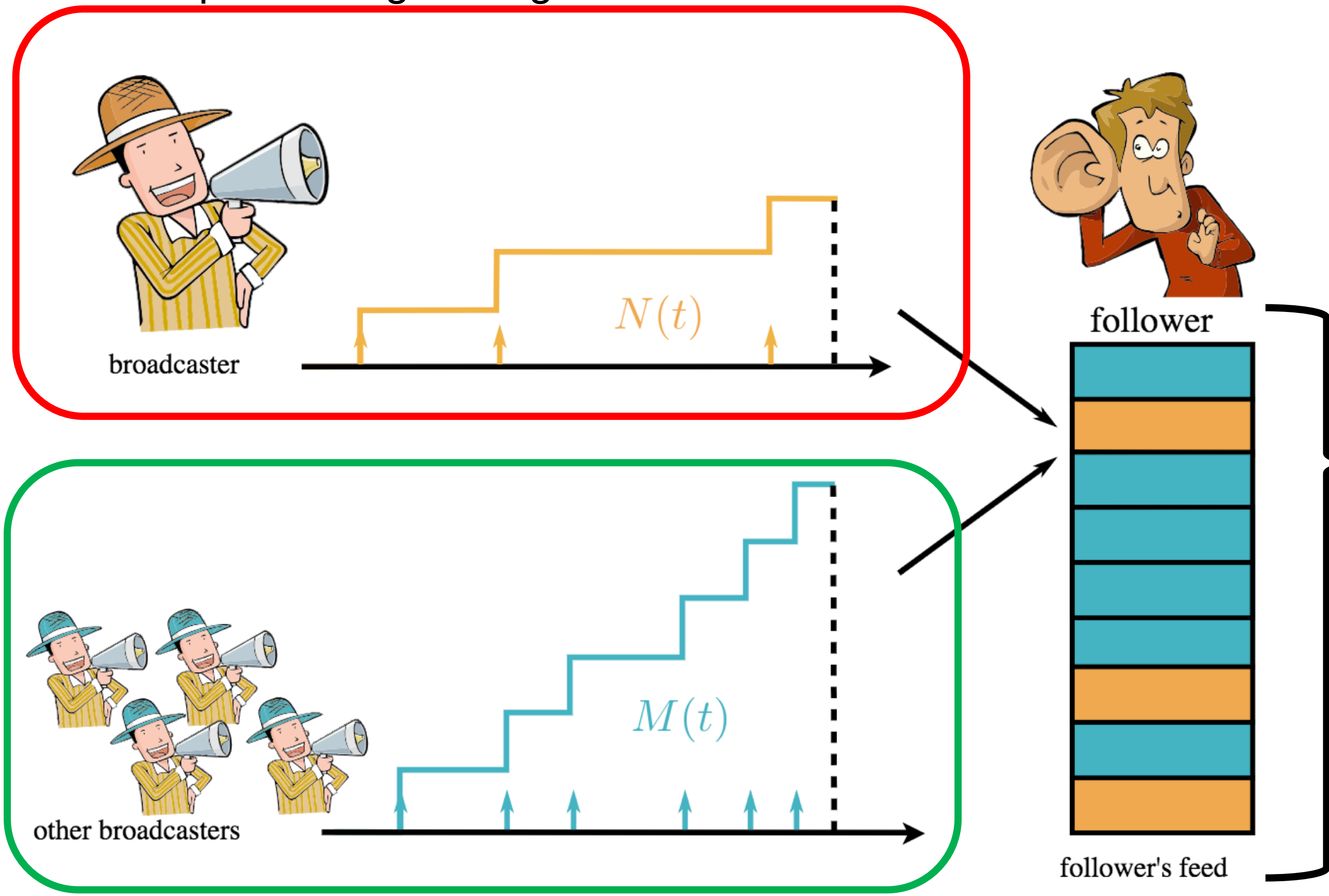
Maximize time spent at the top:

$$\text{maximize} \int_0^T \mathbb{I}(r(t) < 1)\, dt$$

$$\text{s.t.} \int_0^T \lambda(t) \leq C$$

Today: Evaluating broadcasting strategies

Task: Implementing strategies



Simulated

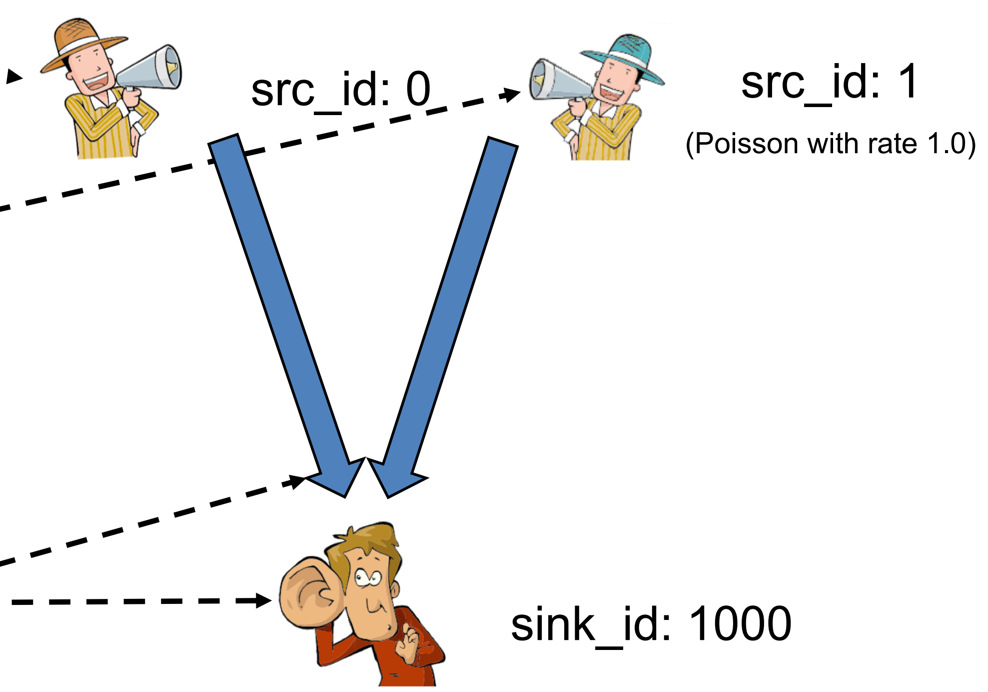
Metrics:

- Average rank
- Time at the top

Working of the simulator

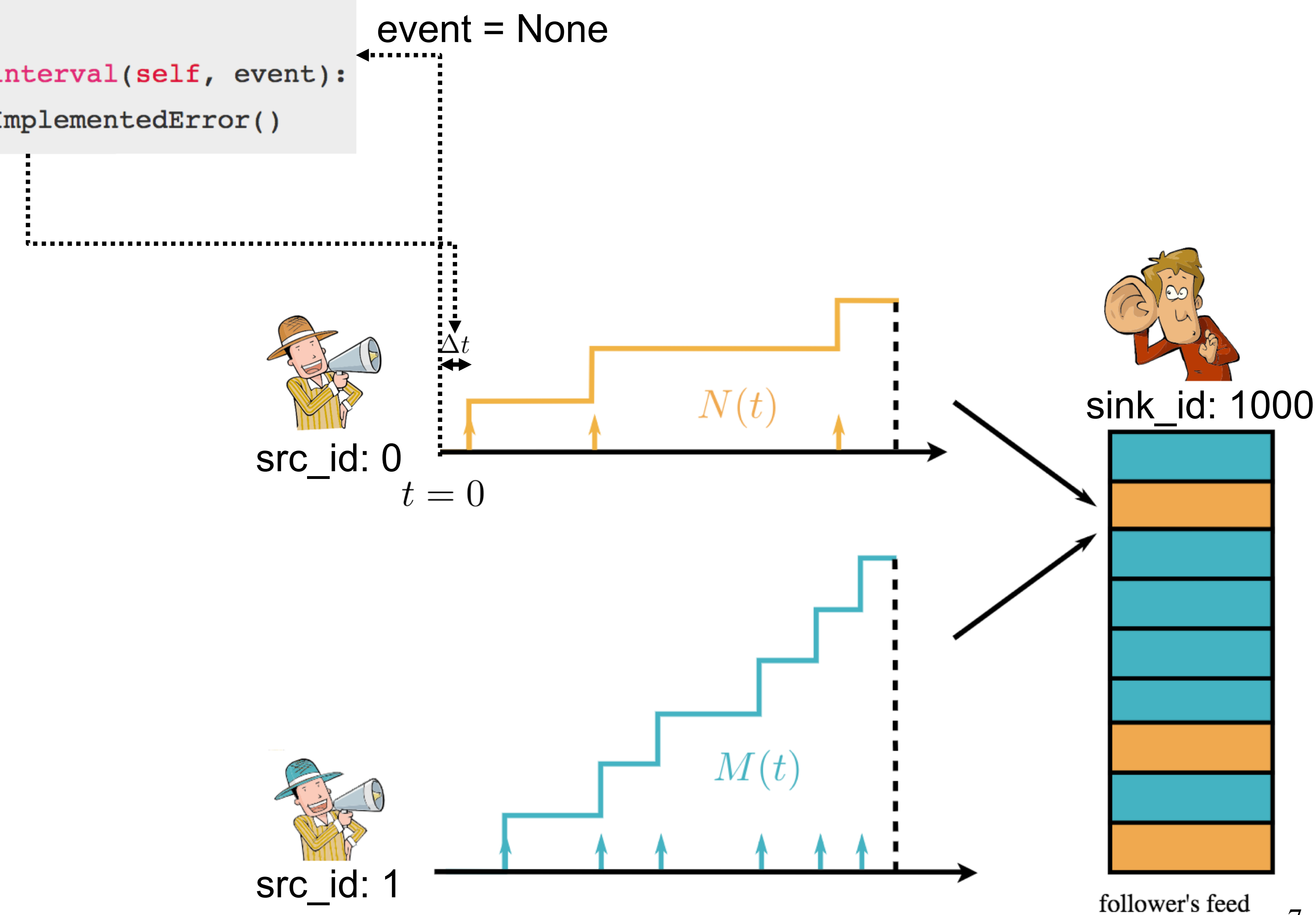
```
sim_manager = opts.create_manager_with_opt(seed)
sim_manager = opts.create_manager_with_poisson(seed, rate)
sim_manager = opts.create_manager_with_smart_poisson(seed, rate)
# ...

sim_manager.run_dynamic()
df = opt_mgr.state.get_dataframe()
# Calculate metrics ...
```

```
Simulation_options = SimOpts(
    # ...
    src_id=0,
    end_time=100,
    other_sources=[
        (
            'Poisson',
            {
                'src_id': 1,
                'seed': 10016,
                'rate': 1.0
            }
        )
    ],
    sink_ids=[1000],
    edge_list=[(0, 1000), (1, 1000)]
)
```



How to *implement* a strategy?

```python
class Broadcaster:
    # ...
    def get_next_interval(self, event):
        raise NotImplementedError()
```

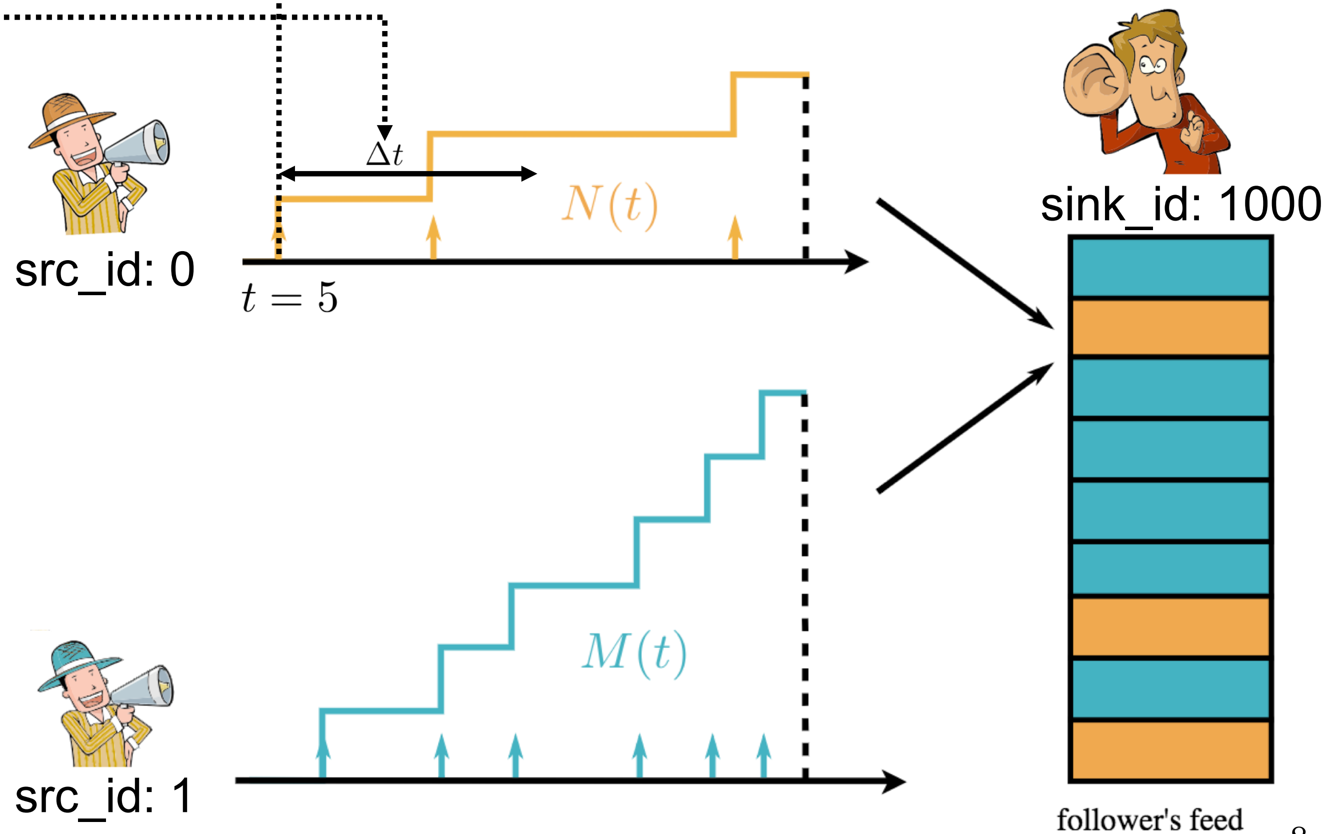


How to *implement* a strategy?

```python
class Broadcaster:
    # ...
    def get_next_interval(self, event):
        raise NotImplementedError()
```

event = Event(event_id=1, time_delta=5, cur_time=5,
src_id=0, sink_ids=[1000])

```python
class Event:
    def __init__(self, event_id, time_delta, cur_time,
                 src_id, sink_ids, metadata=None):
        self.event_id   = event_id
        self.time_delta = time_delta  # Since last event
        self.cur_time   = cur_time
        self.src_id     = src_id
        self.sink_ids   = sink_ids
        self.metadata   = metadata
```

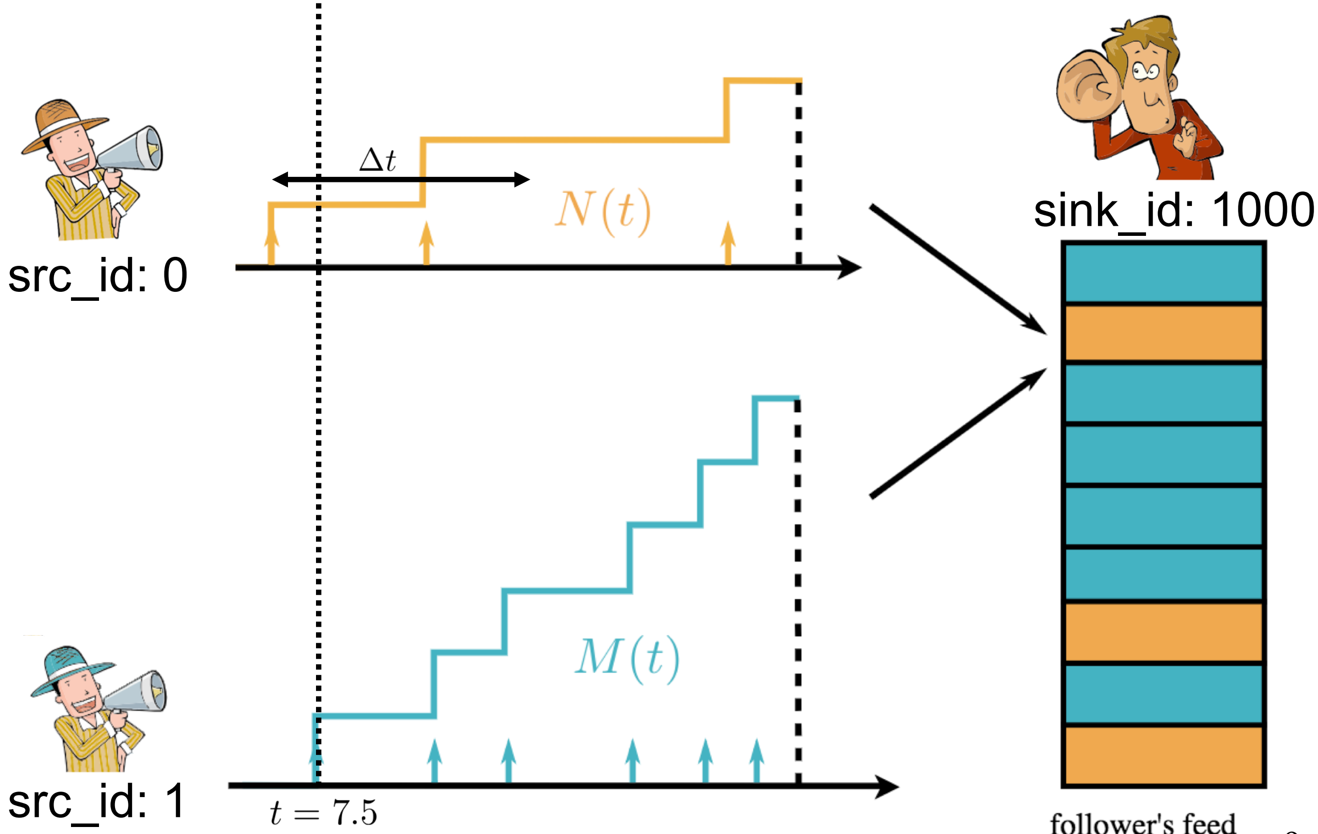


How to *implement* a strategy?

```python
class Broadcaster:
    # ...
    def get_next_interval(self, event):
        raise NotImplementedError()
```

event = Event(event_id=2, time_delta=2.5, cur_time=7.5, src_id=1, sink_ids=[1000])

```python
class Event:
    def __init__(self, event_id, time_delta, cur_time,
                 src_id, sink_ids, metadata=None):
        self.event_id   = event_id
        self.time_delta = time_delta  # Since last event
        self.cur_time   = cur_time
        self.src_id     = src_id
        self.sink_ids   = sink_ids
        self.metadata   = metadata
```



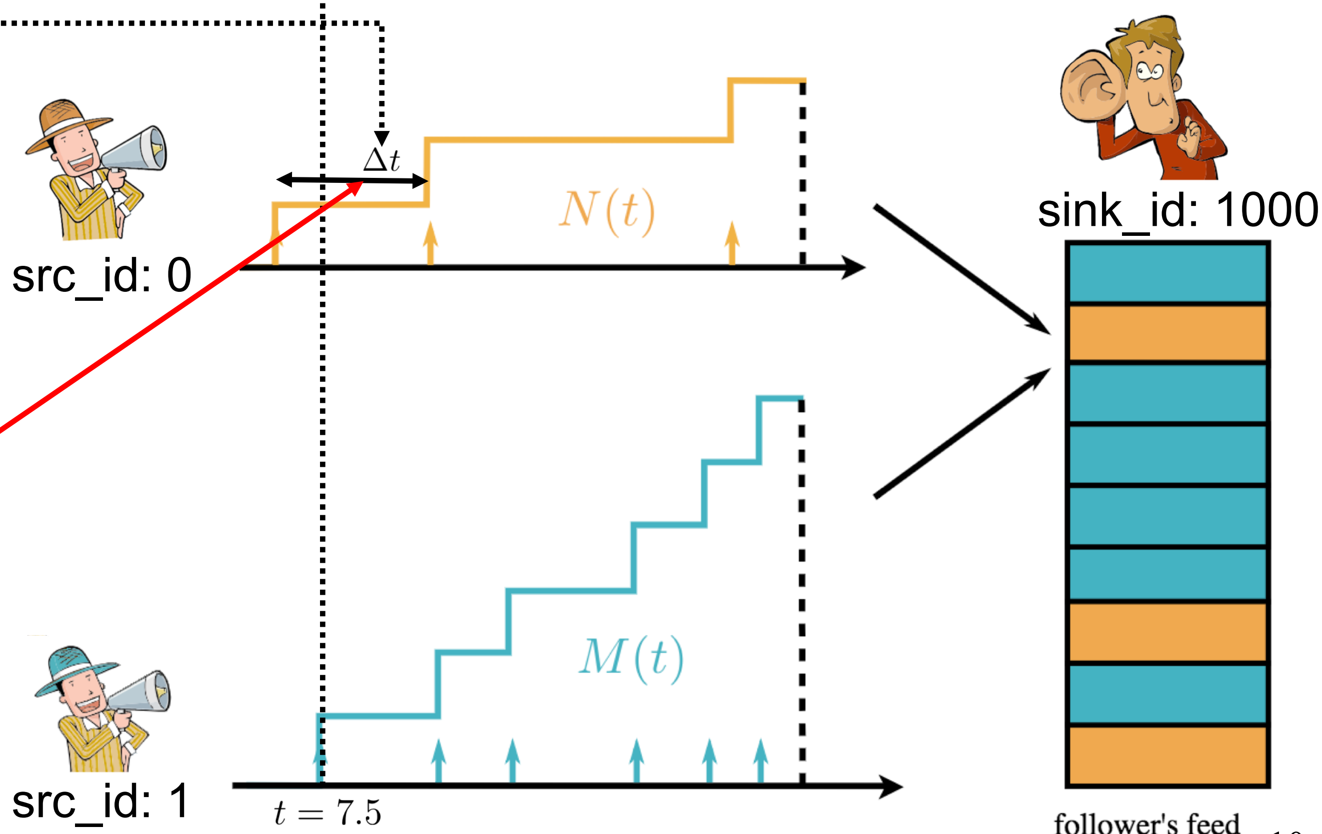
How to *implement* a strategy?

```python
class Broadcaster:
    # ...
    def get_next_interval(self, event):
        raise NotImplementedError()
```

event = Event(event_id=2, time_delta=2.5, cur_time=7.5, src_id=1, sink_ids=[1000])

```python
class Event:
    def __init__(self, event_id, time_delta, cur_time,
                 src_id, sink_ids, metadata=None):
        self.event_id   = event_id
        self.time_delta = time_delta  # Since last event
        self.cur_time   = cur_time
        self.src_id     = src_id
        self.sink_ids   = sink_ids
        self.metadata   = metadata
```

Δt Updated!

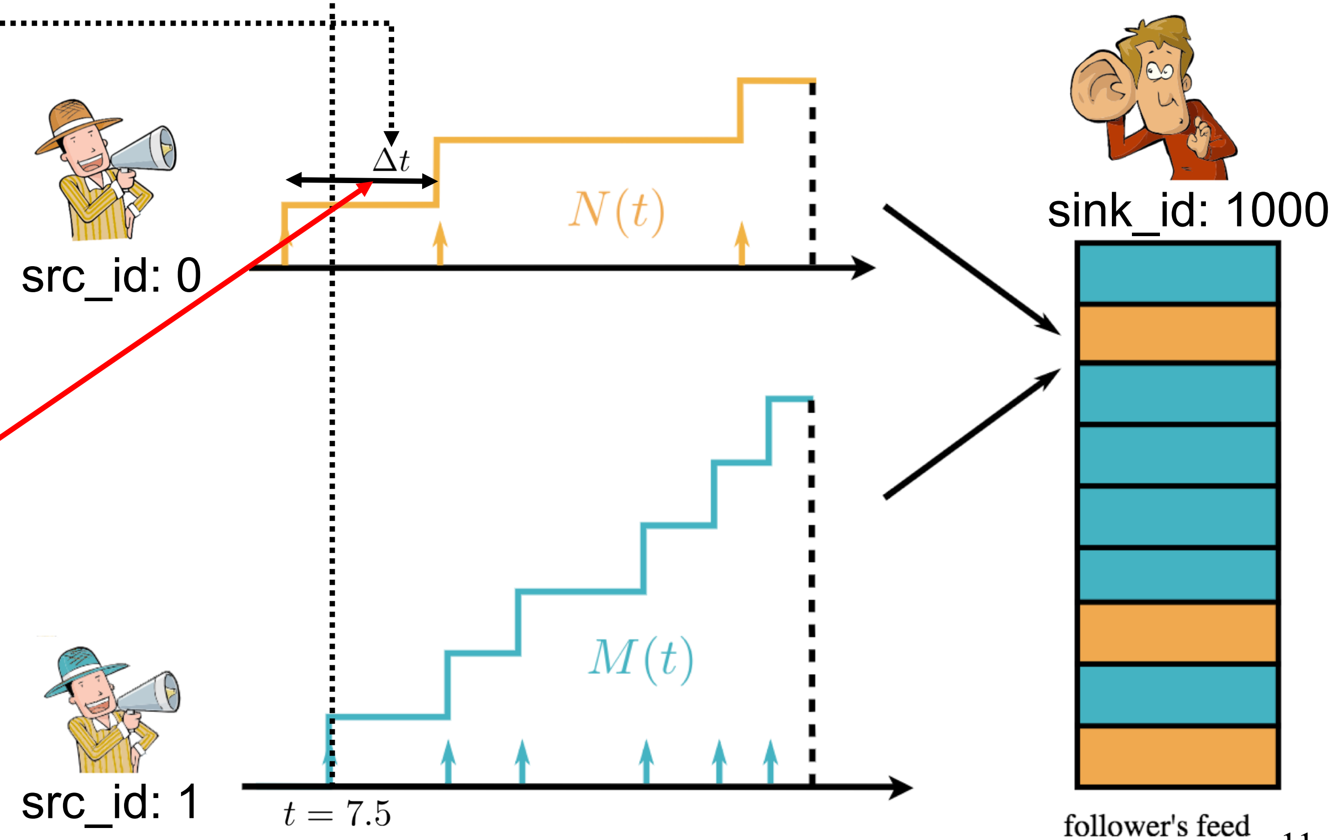


How to *implement* a strategy?

```python
class Broadcaster:
    # ...
    def get_next_interval(self, event):
        raise NotImplementedError()
```

event = Event(event_id=2, time_delta=2.5, cur_time=7.5,
src_id=1, sink_ids=[1000])

```python
class Event:
    def __init__(self, event_id, time_delta, cur_time,
                 src_id, sink_ids, metadata=None):
        self.event_id   = event_id
        self.time_delta = time_delta  # Since last event
        self.cur_time   = cur_time
        self.src_id     = src_id
        self.sink_ids   = sink_ids
        self.metadata   = metadata
```



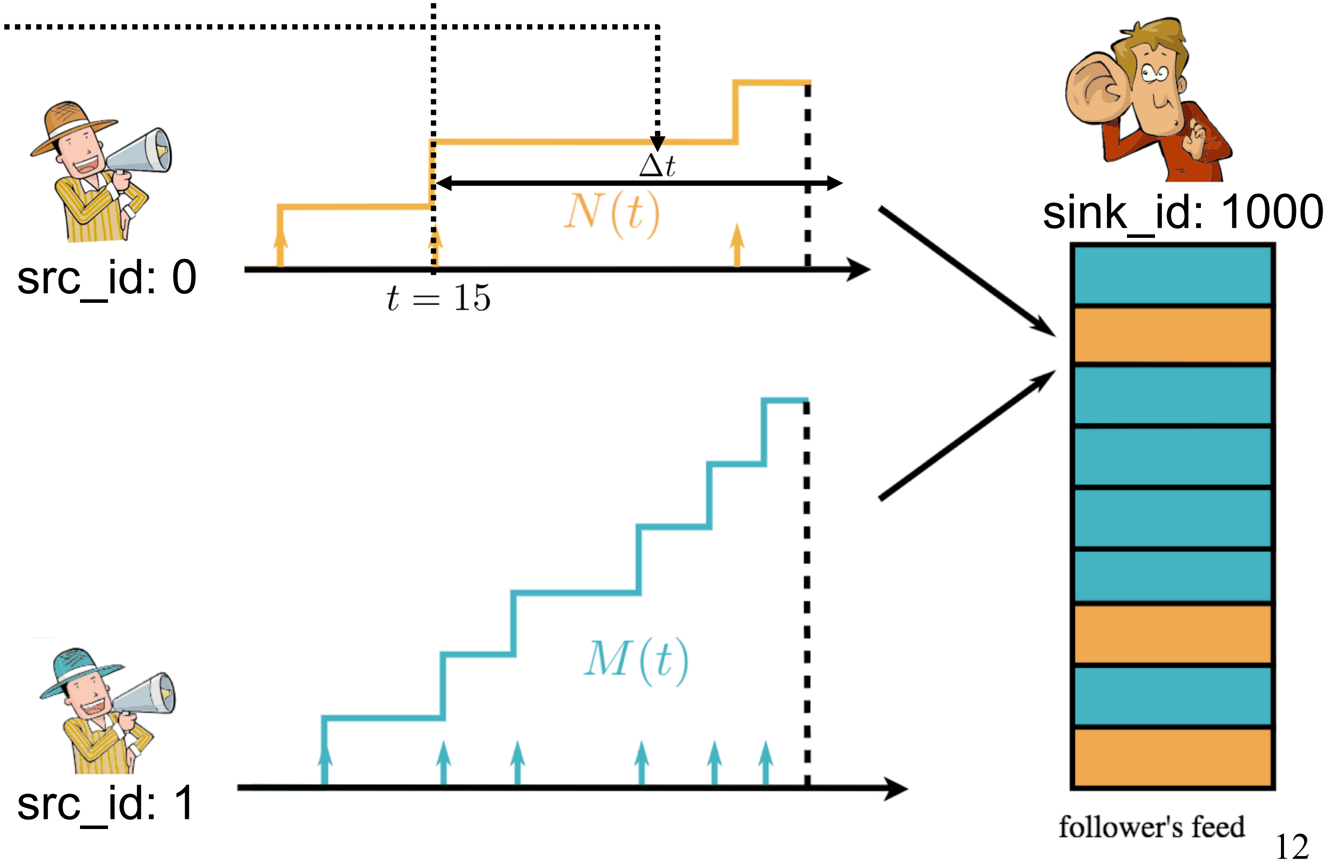
Δt always measured from last *self* post.

How to *implement* a strategy?

```python
class Broadcaster:
    # ...
    def get_next_interval(self, event):
        raise NotImplementedError()
```

event = Event(event_id=3, time_delta=7.5, cur_time=15,
src_id=1, sink_ids=[1000])

```python
class Event:
    def __init__(self, event_id, time_delta, cur_time,
                 src_id, sink_ids, metadata=None):
        self.event_id   = event_id
        self.time_delta = time_delta  # Since last event
        self.cur_time   = cur_time
        self.src_id     = src_id
        self.sink_ids   = sink_ids
        self.metadata   = metadata
```



Repeats to the end.

Broadcasting Strategies

1. **Poisson** — Already implemented.

   $$\lambda(t) = \mu$$

2. **Hawkes** — This lecture.

   $$\lambda(t) = \mu + \alpha \sum_{t_i \in \mathcal{H}(t)} \exp\left(-\beta(t - t_i)\right)$$

3. **RedQueen** — To implement.

   $$\lambda(t) = c\, r(t)$$

4. ***Smart Poisson*** — To implement.

   $$\lambda(t) = \mu\, \mathbb{I}(r(t) > 0)$$

Poisson broadcaster: already implemented

$$\lambda(t) = \mu$$

```python
class Poisson(Broadcaster):
    def __init__(self, src_id, seed, rate=1.0):
        super(Poisson, self).__init__(src_id, seed)
        self.rate = rate

    def get_next_interval(self, event):
        RS = self.random_state
        if event is None or event.src_id == self.src_id:
            # Draw a new time, one event at a time
            scale = 1.0 / self.rate
            return RS.exponential(scale)
```

Poisson broadcaster: already implemented

$$\lambda(t) = \mu$$

Use *self.random_state* for repeatable experiments and debugging.

```
class Poisson(Broadcaster):
    def __init__(self, src_id, seed, rate=1.0):
        super(Poisson, self).__init__(src_id, seed)
        self.rate = rate

    def get_next_interval(self, event):
        RS = self.random_state
        if event is None or event.src_id == self.src_id:
            # Draw a new time, one event at a time
            scale = 1.0 / self.rate
            return RS.exponential(scale)
```

If this is the beginning of the simulation *or* if the post was by this broadcaster, return a new sample.

No *else* branch: not returning a value means *do not* change old time.

Hawkes broadcaster: another example

$$\lambda(t) = \mu + \alpha \sum_{t_i \in \mathcal{H}(t)} \exp\left(-\beta(t - t_i)\right)$$

```python
class Hawkes(Broadcaster):
    def __init__(self, src_id, seed, l_0=1.0, alpha=1.0, beta=10.0):
        super(Hawkes, self).__init__(src_id, seed)
        self.l_0   = l_0
        self.alpha = alpha
        self.beta  = beta
        self.prev_excitations = []

    def get_rate(self, t):
        """Returns the rate of current Hawkes at time `t`."""
        return self.l_0 + \
            self.alpha * sum(np.exp([self.beta * -1.0 * (t - s)
                                     for s in self.prev_excitations
                                     if s <= t]))

    def get_next_interval(self, event):
        t = self.get_current_time(event)
        RS = self.random_state
        if event is None or event.src_id == self.src_id:
            rate_bound = self.get_rate(t)

            # Ogata sampling for one t-delta
            while True:
                t_delta = RS.exponential(scale=1.0 / rate_bound)

                # Rejection sampling
                if RS.rand() < self.get_rate(t + t_delta) / rate_bound:
                    break
                else:
                    t += t_delta

            self.prev_excitations.append(t + t_delta)
            return t + t_delta - self.get_current_time(event)
```

Initializing and saving parameters

Calculating $\lambda(t^+)$

Ignore unless it was our event or the 1st event

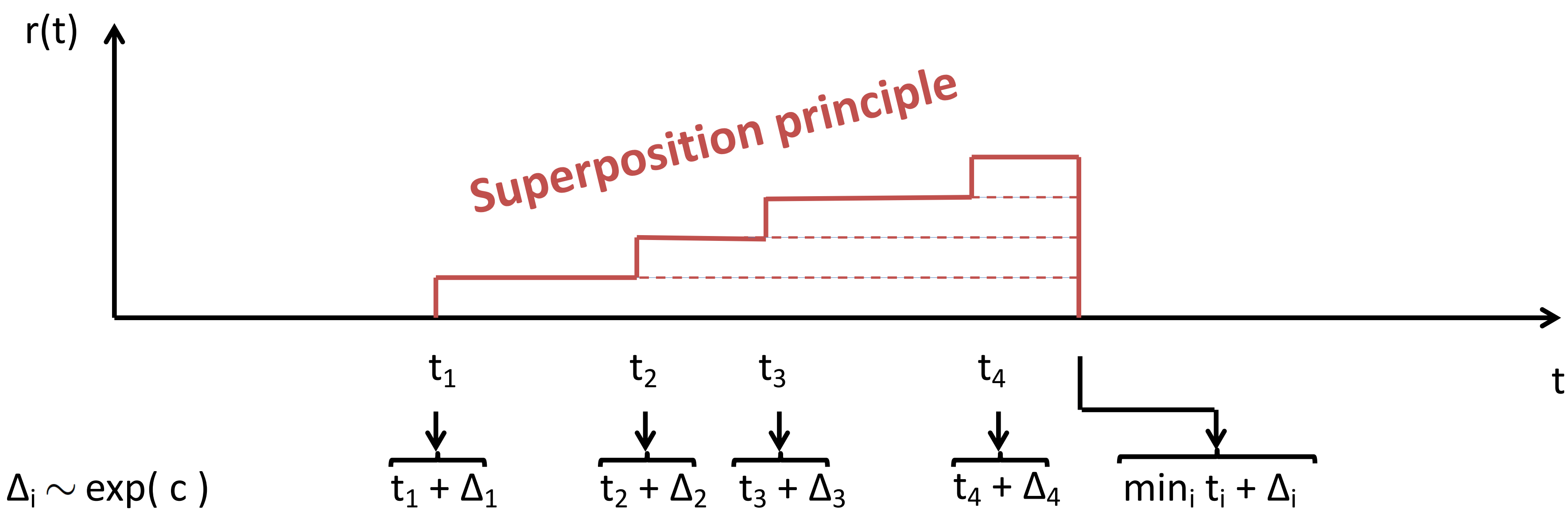
Ogata's thinning algorithm

Return Δt from our own post

Recap: RedQueen broadcaster

$$\lambda(t) = c\, r(t)$$

- Minimizes loss: $\int_0^T \left(r^2(t) + c\,\lambda^2(t) \right) \, dt$
- For the task: $c = \sqrt{\frac{s}{q}} = 1$
- Sampling using Superposition:



RedQueen Broadcaster implementation

$$\lambda(t) = c\, r(t)$$

```python
class Opt(Broadcaster):
    def __init__(self, src_id, seed, q=1.0, s=1.0):
        super(Opt, self).__init__(src_id, seed)
        # ...
        self.rank = 0

    def get_next_interval(self, event):
        if event is None:
            # Tweet immediately if this is the first event.
            self.rank = 0
            return 0
        elif event.src_id == self.src_id:
            # No need to tweet if we are on top of all walls
            self.rank = 0
            return np.inf
        else:
            # Calculate current rank
            self.rank = self.rank + 1
            # cur_time = self.get_current_time(event)

            # t_delta = ...
            # if ...:
            #     return t_delta
```

RedQueen Broadcaster implementation

$$\lambda(t) = c\, r(t)$$

```python
class Opt(Broadcaster):
    def __init__(self, src_id, seed, q=1.0, s=1.0):
        super(Opt, self).__init__(src_id, seed)
        # ...
        self.rank = 0

    def get_next_interval(self, event):
        if event is None:
            # Tweet immediately if this is the first event.
            self.rank = 0
            return 0
        elif event.src_id == self.src_id:
            # No need to tweet if we are on top of all walls
            self.rank = 0
            return np.inf
        else:
            # Calculate current rank
            self.rank = self.rank + 1
            # cur_time = self.get_current_time(event)

            # t_delta = ...
            # if ...:
            #     return t_delta
```

This is how rank evolves.

RedQueen Broadcaster implementation

$$\lambda(t) = c\,r(t)$$

```python
class Opt(Broadcaster):
    def __init__(self, src_id, seed, q=1.0, s=1.0):
        super(Opt, self).__init__(src_id, seed)
        # ...
        self.rank = 0

    def get_next_interval(self, event):
        if event is None:
            # Tweet immediately if this is the first event.
            self.rank = 0
            return 0
        elif event.src_id == self.src_id:
            # No need to tweet if we are on top of all walls
            self.rank = 0
            return np.inf
        else:
            # Calculate current rank
            self.rank = self.rank + 1
            # cur_time = self.get_current_time(event)

            # t_delta = ...
            # if ...:
            #     return t_delta
```

Return infinite if we do not plan to post.

RedQueen Broadcaster implementation

$$\lambda(t) = c\,r(t)$$

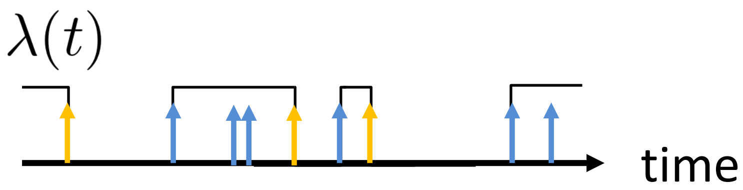
```
class Opt(Broadcaster):
    def __init__(self, src_id, seed, q=1.0, s=1.0):
        super(Opt, self).__init__(src_id, seed)
        # ...
        self.rank = 0

    def get_next_interval(self, event):
        if event is None:
            # Tweet immediately if this is the first event.
            self.rank = 0
            return 0
        elif event.src_id == self.src_id:
            # No need to tweet if we are on top of all walls
            self.rank = 0
            return np.inf
        else:
            # Calculate current rank
            self.rank = self.rank + 1
            # cur_time = self.get_current_time(event)

            # t_delta = ...
            # if ...:
            #     return t_delta
```

$\Delta_i \sim \exp(\ 1.0\ )$

$\min_i t_i + \Delta_i$

Smarter than Poisson Broadcaster

$$\lambda(t) = \mu \, \mathbb{I}(r(t) > 0)$$



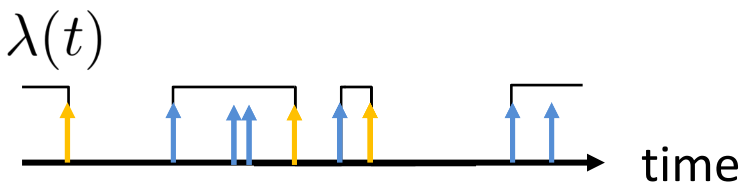

```python
class SmartPoisson(Broadcaster):
    """Like the Poisson Broadcaster,
    but does not post if already on top."""

    def __init__(self, src_id, seed, rate=1.0):
        super(SmartPoisson, self).__init__(src_id, seed)
        self.is_dynamic = True
        self.rate = rate
        self.on_top = False

    def get_next_interval(self, event):
        RS = self.random_state
        if event is None:
            return NotImplemented
        elif event.src_id == self.src_id:
            self.on_top = True
            return np.inf
        elif self.on_top:
            # If we are no longer on top, schedule a post.
            self.on_top = False
            return NotImplemented
```

Heuristic to improve time at top:

- Do not post if already on top.
- If not on top, then post at a steady pace to let *bursts* of others' posts pass (*e.g.*, breaking news).
- Contrast: always maintaining low rank.

Smarter than Poisson Broadcaster

$$\lambda(t) = \mu\,\mathbb{I}(r(t) > 0)$$

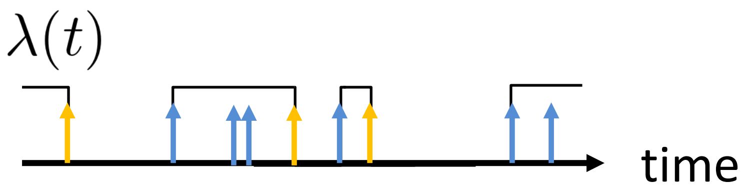



```python
class SmartPoisson(Broadcaster):
    """Like the Poisson Broadcaster,
    but does not post if already on top."""

    def __init__(self, src_id, seed, rate=1.0):
        super(SmartPoisson, self).__init__(src_id, seed)
        self.is_dynamic = True
        self.rate = rate
        self.on_top = False

    def get_next_interval(self, event):
        RS = self.random_state
        if event is None:
            return NotImplemented
        elif event.src_id == self.src_id:
            self.on_top = True
            return np.inf
        elif self.on_top:
            # If we are no longer on top, schedule a post.
            self.on_top = False
            return NotImplemented
```

Using a flag to figure out if on top or not.

Smarter than Poisson Broadcaster

$$\lambda(t) = \mu\, \mathbb{I}(r(t) > 0)$$

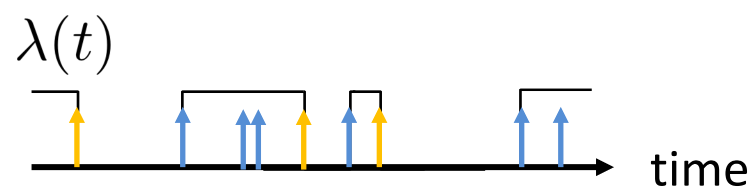



```python
class SmartPoisson(Broadcaster):
    """Like the Poisson Broadcaster,
    but does not post if already on top."""

    def __init__(self, src_id, seed, rate=1.0):
        super(SmartPoisson, self).__init__(src_id, seed)
        self.is_dynamic = True
        self.rate = rate
        self.on_top = False

    def get_next_interval(self, event):
        RS = self.random_state
        if event is None:
            return NotImplemented
        elif event.src_id == self.src_id:
            self.on_top = True
            return np.inf
        elif self.on_top:
            # If we are no longer on top, schedule a post.
            self.on_top = False
            return NotImplemented
```

Return infinite if we do not plan to post.

Smarter than Poisson Broadcaster

$$\lambda(t) = \mu\, \mathbb{I}(r(t) > 0)$$




```
class SmartPoisson(Broadcaster):
    """Like the Poisson Broadcaster,
    but does not post if already on top."""

    def __init__(self, src_id, seed, rate=1.0):
        super(SmartPoisson, self).__init__(src_id, seed)
        self.is_dynamic = True
        self.rate = rate
        self.on_top = False

    def get_next_interval(self, event):
        RS = self.random_state
        if event is None:
            return NotImplemented
        elif event.src_id == self.src_id:
            self.on_top = True
            return np.inf
        elif self.on_top:
            # If we are no longer on top, schedule a post.
            self.on_top = False
            return NotImplemented
```

To be implemented.

Live Coding

- Show execution of simulation
- Diagnostic plots
- Evaluation metrics

Evaluation

	RedQueen	Smart Poisson
Top-1	57 ± 3	58 ± 4
Average rank	59 ± 6	67 ± 10
Number of posts	61 ± 3	62 ± 4

Happy coding!

Questions?

- Drop me an e-mail at `utkarshu@mpi-sws.org`
- Skype: `utkarsh.upadhyay`