Emerging Communication Patterns in Actor Systems

Dominik Charousset
dominik.charousset@haw-hamburg.de

iNET RG, Department of Computer Science
Hamburg University of Applied Sciences

November 2014

Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Agenda

Fallacies of Distributed Computing

New developers often have a set of assumptions (L. P. Deutsch):

- The network is reliable
- Latency is zero
- Bandwidth is infinite
- The network is secure
- Topology doesn't change
- There is one administrator
- Transport cost is zero
- The network is homogeneous

These assumptions ultimately prove false, leading to performance issues and system failure.

Fallacies of Actor Programming

Some of Deutsch's fallacies also apply to actor programming:

- Latency is zero
- Bandwidth is infinite
- Topology doesn't change
- There is one administrator
- Transport cost is zero

More Actor Fallacies

There are also new ones:

- Unbound mailboxes means infinite space
- The runtime is intelligent

The Truth about Message Passing

All messages go through ...

The Truth about Message Passing

All messages go through ...

... the river of slime

The Slime

- Actors drop messages into the slime
- Other actors eventually take it out
- The slime is not intelligent
- The slime fights back if you overload it

Angering the Slime

How to anger the slime:

- Send large messages to compute very little
- Add many, many buffers by building complex topologies
- Send messages faster than they are consumed

If you do any of this, the slime will slow you down or even kill you.

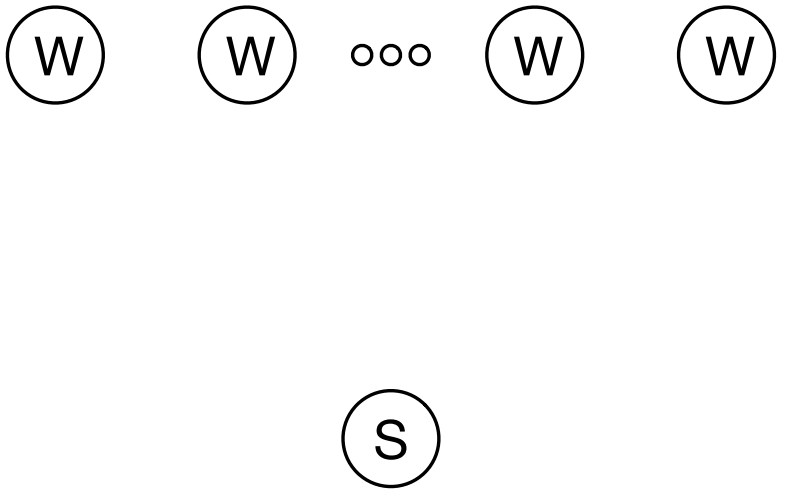
Agenda

Causes for Overloads

Different communication patterns cause different overload scenarios

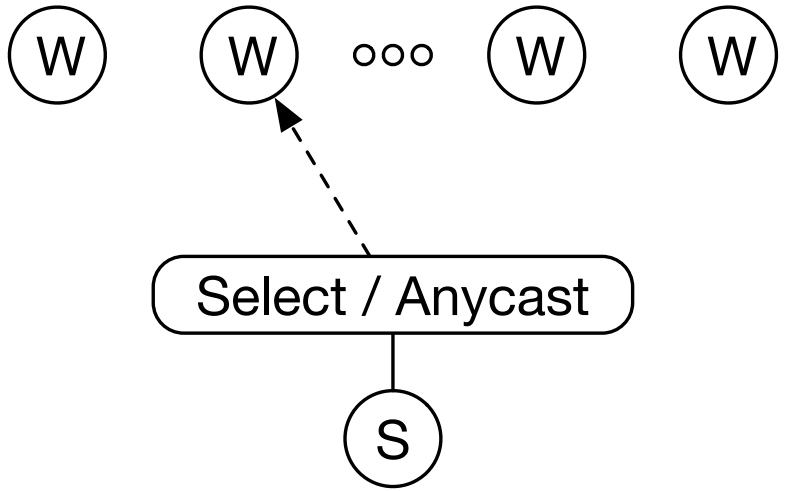
- External systems send too many requests, all nodes overloaded
- Single dispatcher (1:N) acts as bottleneck, others under-utilized
- Dispatcher (1:N) has no feedback channel, all workers overloaded
- (Accidental) N:1 message flows shuts down a single node

1:N Patterns



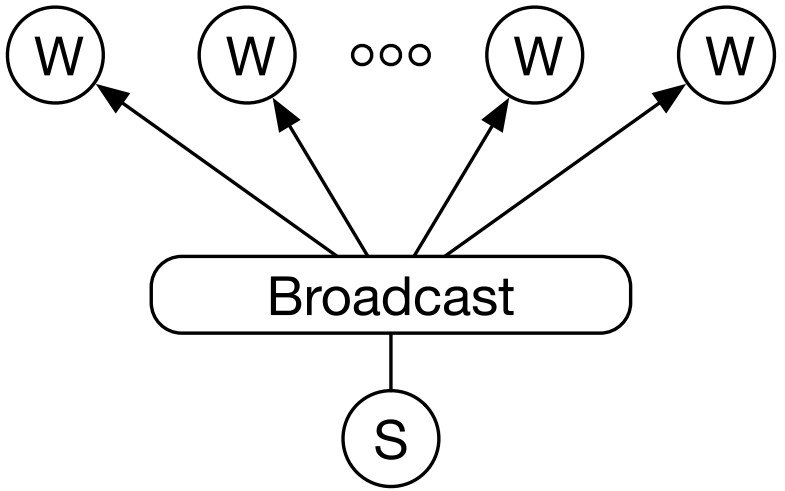
- 1 Supervisor (S)
- N Workers (W)

Select Single Worker



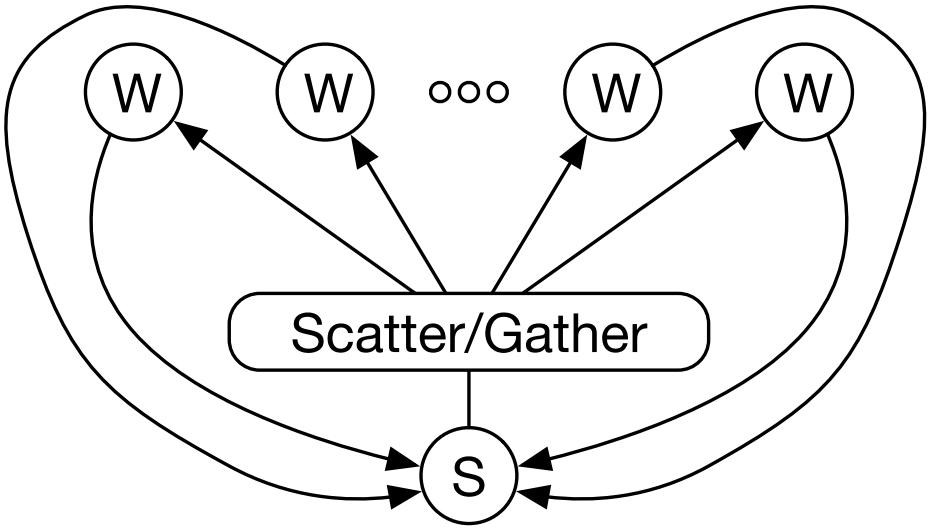
- Supervisor delegates single work items
- Strategy is critical (round-robin, shortest mailbox, ...)

Broadcast



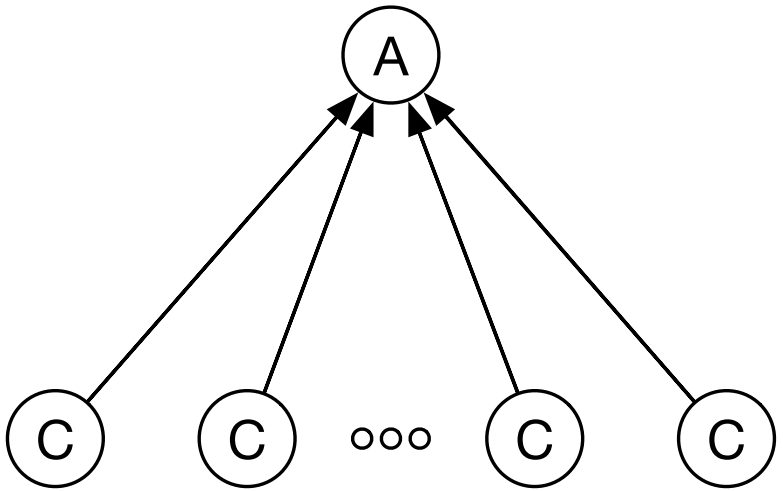
- Supervisor multiplies work items
- Fair scheduling among workers required

Scatter/Gather



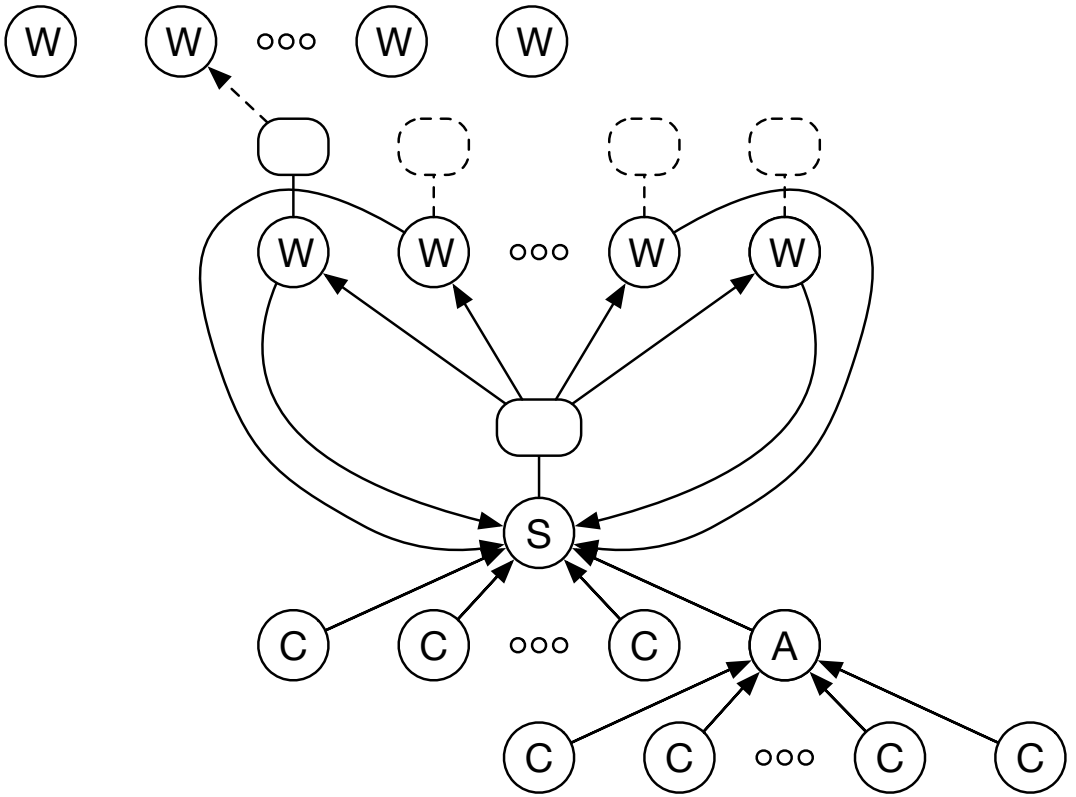
- Supervisor multiplies work items & collects/combines results
- Same as broadcast + heavy load on Supervisor

N:1 Patterns



- Single actor (A) offering a service
- N clients (C) using it

N:M Compositions



- Real-world systems are compositions of several patterns
- Complex service hierarchies, patterns emerge during development

Load Management

- Actors must manage the flow of messages
- Actors in critical path must be “defended” against overload
- Feedback channel between receiver and sender(s) needed
- Senders must adopt to consumption rate

Agenda

Preliminary Considerations

- Messaging layer cannot “stop” actors from sending (too much)
- Slime is not intelligent, actors are (should be)
- Latency cannot be foreseen or precalculated, only observed
- Bursts must not stop the system from running
- Silently dropping messages not an option
  - Errors have to be explicit
  - Dropping messages *is* a valid strategy, if observable

Goals

- Make emerged communication patterns explicit
  - Traceable interaction between identifiable subsystems
  - Ease reasoning about software architecture

- Provide high-level software building blocks
  - "Standardized" message types for composable interfaces
  - Simple setup for common patterns, configurable and tunable

Architecture Design Space

- "Bottom-Up": Build each subsystem individually as black box
  - Hides service hierarchies
  - No inter-component dependencies
  - But: overhead (load management) between all components

- "Top-Down": Build topology for large (sub-) system
  - Service hierarchies explicitly modeled
  - Load management only once at each critical path
  - But: (potentially) high complexity

Load Management

- Do as much work as possible, but not more
- Buffers can become harmful: system should stay responsive
- Reject task as soon as possible, before committing resources
- Once task is accepted, complete as early as possible
- Give clients clear indication why task has been rejected
  - User might try again later (temporary burst)
  - Allow non-interactive client to slow down or propagate error

Management Strategies

- Token Bucket
  - Each task consumes a token
  - One token per worker + buffer for small bursts
  - Finished tasks release token
  - Reject task if no token is left
- Rate Limit
  - Allow only X tasks/second
  - Based on engineered capacity or adaptive
- Wait queue with limited sojourn time
  - Sojourn time: time between enqueue & dequeue
  - Restrict time elements are allowed to stay in queue
  - Reject task if no worker became available in due time
  - Defines worst case latency

Load Management Design Space

- Static vs. dynamic setup (templated vs. virtual dispatching)
  - Templated code makes algorithm choice explicit
  - Virtual dispatching allows for runtime configuration
- Feedback loop design (negative vs. positive)
  - Small buffer at worker can minimize idle time
  - No buffer at worker avoids delaying tasks unnecessarily
- Buffer sizes and adaption rates
  - Balance between additional latency and short burst smoothening
  - Configurable rates require good guidance for users

Agenda

Erlang: “Jobs” Load Regulation Framework

- Regulation at the edges
- Highly configurable load management framework
- Protects core nodes from overloads by rejecting excessive work
- Inter-node feedbacks, pluggable queues, etc.
- Subsystem as black box

Akka: Routers

- Route incoming messages to outbound actors
- Several predefined strategies:
  - RoundRobinRoutingLogic
  - RandomRoutingLogic
  - SmallestMailboxRoutingLogic
  - BroadcastRoutingLogic
  - ScatterGatherFirstCompletedRoutingLogic
  - ConsistentHashingRoutingLogic
- Serve as building block for load management

Agenda

1. Fallacies of Actor Messaging
2. Overload Scenarios & Communication Patterns
3. Goals & Design Space
4. Example Frameworks
5. Conclusion

Conclusion

- Communication patterns should be explicitly designed
- Understanding message flow is crucial for performance
- Ideal position of load managers depends on patterns

Thank you for your attention!

Feedback and ideas?