Run Actor Run - ChatApp benchmark

ChatApp models a chat application in the spirit of e.g., Facebook chats. It consists of three types of application actors: *clients*, *directories* and *chats*. A client is a server-side proxy for an imagined client on the client-side. A directory is a load-balancing mechanism that maps a number of client ids to their corresponding actor handles. A chat is a representation of a conversation between multiple client actors that keeps a history of a conversation, and a list of clients involved in the chat to which it forwards all incoming chat messages.

The relationship between these actors is as follows:

- A *client* may have several friends (other clients).
- Each *client* belongs to exactly one *directory*, and can only be reached through this *directory* or by one of his friends directly.
- A *client* may participate in several *chats*.
- Participation of clients in chats is not constant,
- Friendship is constant, but tuneable (see *befriend* below).

Ideally, for running the benchmark, there should be an "external world" actor to supply non-random, repeatable input to the system. For that purpose, we employ a single *poker* actor to stream events into the system deterministically.

To ensure comparability between implementations and take the effects of potential virtual machine warm-up into account, the benchmark operates in *iterations*. An iteration is full end-to-end interaction with the chat system. At the start of the benchmark, a new poker is created which is re-used for every iteration. At the start of an iteration, the *poker* logs-in clients (by ID) at the directories in a round-robin fashion. During login, the directory "arbitrarily" decides to befriend clients. What we mean by "randomly" is discussed later in this section.

After the login phase, the poker operates in *turns*. More specifically, the poker is used to saturate the system with "external events" and we refer to these events as turns. Consequently, the poker sends asynchronous "*poke"* messages to every directory:

```
for t in Range[U64](0, _turns) do
  for directory in _directories.values() do
    //the very first poke message starts turn $T_J$ of iteration $I_I$
    directory.poke(...)
  end
end
```

Upon receipt of such a message, a directory tells its clients to *act*:

```
be poke(...) =>
  for client in _client.values() do
    client.act(...)
  end
end
```

Every client then independently picks one of the following tasks and executes it, each of which is assigned to a probability with which it is picked:

1. Post a message to a “randomly” picked chat (*post*).
2. Leave a “randomly” picked chat (*leave*).
3. Execute a stack-recursive fibonacci number computation (*compute*).
4. Invite a subset of the client’s friends to a newly created chat (*invite*).
5. Do nothing.

When all clients have *completed* their task (transitively), a turn is considered to be completed. More specifically,

1. *post*: all clients within the chat to which a message has been posted have acknowledged the receipt of that message.
2. *leave*: the chat has confirmed the receipt of the leave message.
3. *compute*: computation of the fibonacci number has been completed.
4. *invite*: the chat has confirmed the join of the invited subset and this subset has confirmed the receipt of the buffered chat history.
5. *Do nothing:* the receipt of the *act* marks the end of the task.

Note that the poker does not wait for a turn to be completed before submitting the next turn. The non-deterministic nature of actor systems and scheduling allows for turn T_J to complete before turn T_{J+1}. This leads to the following questions:

a) How much concurrency does ChatApp allow?
    i) Clients execute independently and are poked via directories. That is, the number of directories limits the speed of parallel client activations, as directories may be contended for concurrent access.
b) How does actor scheduling affect how much work is being done per turn?
    i) Scheduling can have a dramatic effect on the total number of messages being sent. For example, post messages may be forwarded to fewer or more clients depending on whether clients have been scheduled to join a chat. To mitigate this effect, we replay all buffered messages to any client joining a chat.

c) How does scheduling affect the measurements?
    i) As mentioned above, we use accumulators to measure the time taken for every individual turn. However, since the accumulator is also an actor, it may only stop timing once it has been scheduled. Consequently, the times measured or an upper bound and do not precisely reflect the time required for completing tasks.

d) To which extent is the ChatApp benchmark deterministic?
    i) One of our main objectives is to design a benchmark that produces comparable results across different actor language and runtime libraries. For that purpose, we do not rely on true randomization. ChatApp uses a deterministic, simple congruential number generator with a constant seed to generate the sequence of tasks to be executed. More specifically, each turn and iteration is repeatable between languages and runtimes.

Tuning and Measuring the Benchmark

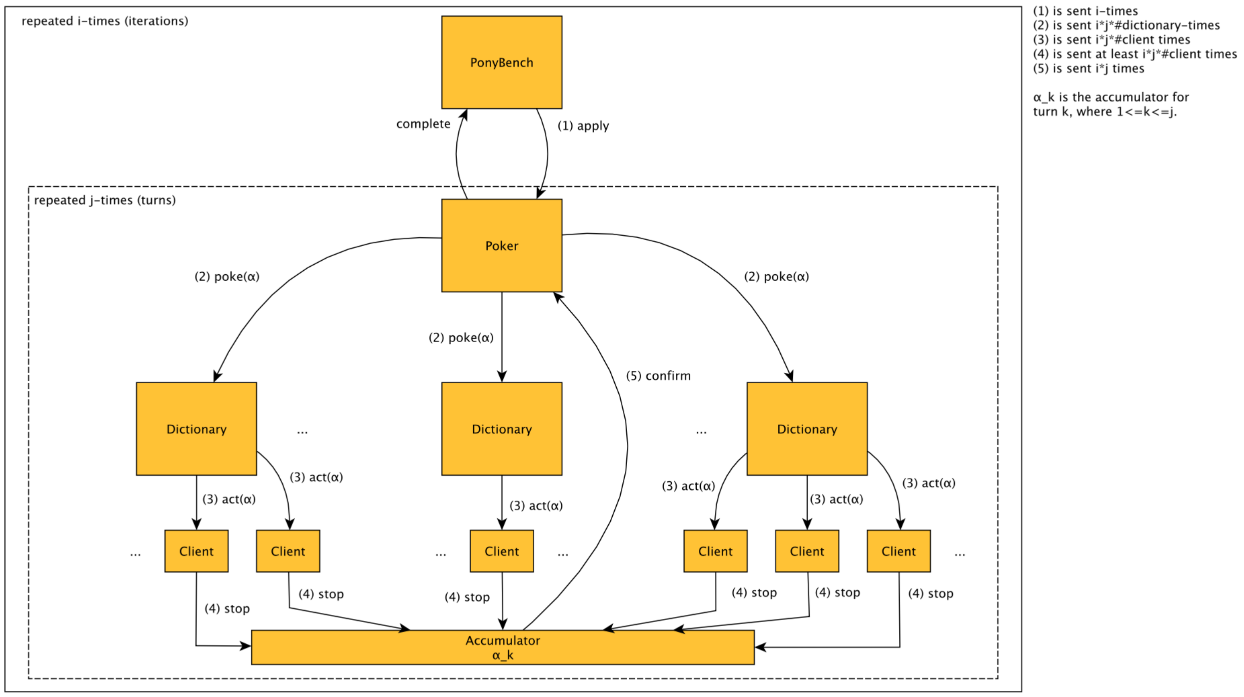
ChatApp is designed to be highly tunable, that is it can be configured using command-line arguments to allow exploring how a language handles different scenarios, e.g. …:

a) how many *turns* should be executed,
b) how many *clients* and *directories* should be spawned and
    i) The latter allows to control the degree of concurrency and parallelism
c) the probabilities for a client to execute either a *compute*, *leave*, *post* or *invite* task.

During one iteration, we measure (1) the time between start and completion of each turn individually as well as (2) the time taken to complete all turns. Namely, (1) is the time from when the poker sends the first *poke* until all pokes from the same turn have been confirmed by all directories, (2) is the time from the first broadcast in the first turn until all turns have been confirmed. (2) can be measured implicitly by measuring a single invocation of ChatApp.

Measuring (1) is intricate because it is possible for a directory to be sending out confirm messages pertaining to turn T_j, while concurrently receiving ack messages pertaining to turns beforeT_{j-1}. To address this, we introduce *accumulator* actors. There is one *accumulator* per turn and iteration: let us call these i and j. When the *accumulator* gets created, it records the time, and so knows when it started. The `act` requests pass an accumulator as an argument. Clients confirm completion of task by sending a stop message to the relevant accumulator. The turn is complete once all accumulators have received as many `stop`s as there are client-s in the turn. It then reports the duration between current time and the recorded start time back to the Poker. This is what we call $T_{\{i,j\}}$ below.

The following diagram shows the message flow of ChatApp:



Between iterations accumulators, clients and chats are eventually deallocated, but the poker and the directories persist. The clients execute the same "script" within every iteration.

We measure in milliseconds

1. The duration of each turn in each iteration, $T_{\{i.j\}}$
2. The duration of each iteration

Based on that, we calculate

3. The average of all turns across all iterations
4. The average of all iterations
5. The standard deviation of the individual timings of the iterations
6. The error window of the individual timings of the iterations
7. The standard deviation of the j-th turn across all iterations (we call this measurement "quality of service")

Note that the duration of an iteration is *not* the sum of the duration of all its turns. This is so, because turns may overlap in time, and also because an iteration includes the deallocation of clients, and actions that happen after all turns have completed.

Benchmarking - Five scenarios

For cross-language benchmark comparisons, we propose 5 configurations to be part of the standard ChatApp benchmark suite.

1. ***Mixed***. Intended to simulate characteristics of message sends and interleaved computations.
   a. `turns = 64, clients = 64536, directories = 512, compute = 75, post = 25, invite = 25, leave = 25, befriend = 10`
2. ***Compute-bound***. Never invite, never post and only execute compute behaviors. Leaving a chat is technically irrelevant.
   a. `turns = 64, clients = 64536, directories = 512, compute = 100, post = 0, invite = 0, leave = 0, befriend = 10`
3. ***Message-bound***. Never compute and never leave chats. Only invite and post messages.
   a. `turns = 64, clients = 64536, directories = 512, compute = 0, post = 95, invite = 5, leave = 0, befriend = 10`
4. ***Artificially-sequential***. Mixed-scenario with a single directory.
   a. `turns = 64, clients = 64536, directories = 1, compute = 75, post = 25, invite = 25, leave = 25, befriend = 10`
5. ***Long-living***. Execute a large number of turns in a message bound-scenario. Picking this to be message-bound is deliberate in order to avoid that the benchmark will become memory-bound.
   a. `turns = 131072, clients = 64, directories = 8, compute = 0, post = 95, invite = 5, leave = 0, befriend = 10`

Scenario[1]	Buffered Chats	No Buffered Chats
Mixed	Median: 1946.81 ms Stddev: 198.871 Qos: 244.343	Median: 1855.66 ms Stddev: 126.909 QoS: 184.189
Compute-bound	Median: 903.466 ms Stddev: 99.5596 QoS: 148.477	-
Message-bound	Median: 1810.89 ms Stddev: 591.309 QoS: 629.144	Median: 2178.13 ms Stddev: 492.434 QoS: 478.638
Artificially-sequential	??	??
Long-living	Median: 4737.29 ms Stddev: 515.067 QoS: 780.47	Median: 4572.27 ms Stddev: 515.944 QoS: 653.288

[1] 12 Iterations, Host machine: Intel(R) Xeon(R) CPU E5-2690 v3 @ 2.60GHz, 24-core HT, two NUMA nodes, 256 GB main memory (DDR3-2133).

Proposed Structure of Paper

Title: Towards principled, tunable, cross actor-language benchmarking

1. *Abstract*: We can essentially use the one from the AGERE paper, but I think we should stress the need for a tunable benchmark.
2. *Introduction*:
    a. Why do we need to benchmark in general?
    b. Why benchmarking actor languages / implementations?
    c. Why a new benchmark suite?
3. *Background research: actor -languages/runtimes and their design goals and differences*
    a. *Pony*
    b. *Akka*
    c. *Erlang*
    d. *CAF*
    e. *...*
4. *Background research: actor -language/-runtime benchmarks*
    a. What are good ideas?
    b. What are flaws?
5. *The "fallacies" of benchmarking*:
    a. What are the pitfalls of cross-language/cross-implementation benchmarking?
        i. Implementation, randomness, setup, measurements, …
6. *The Objective*
    a. What we want to achieve and what our rule set is.
7. *ChatApp - Specification*
8. *ChatApp - Reference Implementation deep-dive (Pony)*
9. *Benchmark setup*
    a. How do we measure?
    b. How do we ensure comparability?
    c. How do we provide confidence that an implementation meets the specification?
10. *Cross-language / -runtime comparisons*
    a. Shortly explain implementation differences due to language/runtime differences (e.g. poison pills)
11. Results
12. Conclusion
13. Future Work