Aneris: A Mechanised Logic for Modular Reasoning about Distributed Systems

Morten Krogh-Jespersen[1] Amin Timany[2]
Marit Edna Ohlenbusch[1] Simon Oddershede Gregersen[1]
Lars Birkedal[1]

[1]Aarhus University, Aarhus, Denmark

[2]imec-DistriNet KU Leuven, Leuven, Belgium

October 28, 2019

Iris workshop,
Aarhus University

Introduction

Distributed systems are ubiquitous

Some applications are critical, e.g., online banking

Hence, there is a need for verification

Efficient implementations often use advanced features like node-local concurrency and higher-order memory

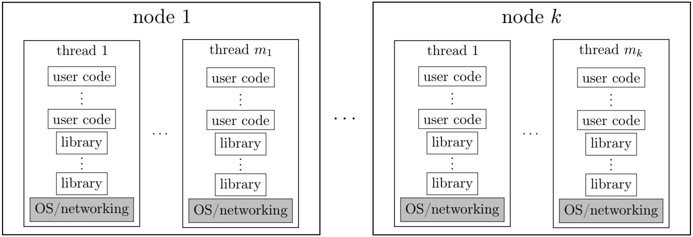
It is well known that reasoning about distributed programs is difficult

Traditionally, most works focus on verifying a high-level model of the system

We introduce Aneris: a program logic for modular verification of safety of distributed systems' code

In this talk

- AnerisLang: an advanced programming language for programming distributed systems
- Aneris logic: a program logic for modular reasoning about AnerisLang programs
  - Horizontal modularity: nodes, and threads, are verified in isolation and the proofs are composed
  - Vertical modularity: library code is verified separately and library clients are verified against the library specs.



In this talk

- AnerisLang: an advanced programming language for programming distributed systems
- Aneris logic: a program logic for modular reasoning about AnerisLang programs
  - Horizontal modularity: nodes, and threads, are verified in isolation and the proofs are composed
  - Vertical modularity: library code is verified separately and library clients are verified against the library specs.
- Concurrent-separation-logic-style specs for distributed systems
- Examples of distributed network specifications

AnerisLang

In this work, we introduce AnerisLang and the Aneris logic.

AnerisLang: an untyped ML-style programming language with

- UDP-like network primitives
    - unordered messages
    - possibility of dropped packets
    - duplicate protection
- Concurrency (multiple threads on each node)
- Higher-order memory (can store code in memory)
- Primitive types and support of (de)serialization to strings
- Can (almost) be directly extracted to running OCaml code

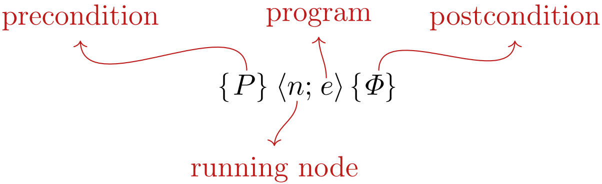
The operational semantics keeps track of

- A heap for each node
- A mapping from socket handles to socket addresses (ip and port) for each node
- A message soup: a collection of sent messages

Aneris logic

Aneris logic: a program logic based on Iris for distributed systems

Hoare triples:



precondition

program

postcondition

$$\{P\}\,\langle n; e\rangle\,\{\Phi\}$$

running node

Hoare triples guarantee safety:

If $\{P\}\,\langle n; e\rangle\,\{\Phi\}$ holds then

- given that P holds for the initial state,
- e is safe, *i.e.* the program (distributed system) will not crash,
- and if it terminates with a final value v, $\Phi(v)$ holds.

Modularity in Aneris logic

- Thread-local reasoning:

$$\frac{\{P_1\}\,\langle n; e_1\rangle\,\{v.Q_1\} \qquad \{P_2\}\,\langle n; e_2\rangle\,\{v.Q_2\}}{\{P_1 * P_2\}\,\langle n; e_1 \mid\mid e_2\rangle\,\{v.\exists v_1, v_2. v = (v_1, v_2) * Q_1[v_1/v] * Q_2[v_2/v]\}}$$

- Node-local reasoning:

$$\frac{\begin{array}{c}\{P_1 * \mathsf{IsNode}(n_1) * \mathsf{FreePorts}(ip_1, \mathfrak{P})\}\,\langle n_1; e_1\rangle\,\{\mathsf{True}\} \\ \{P_2 * \mathsf{IsNode}(n_2) * \mathsf{FreePorts}(ip_2, \mathfrak{P})\}\,\langle n_2; e_2\rangle\,\{\mathsf{True}\}\end{array}}{\{P_1 * P_2 * \mathsf{FreeIp}(ip_1) * \mathsf{FreeIp}(ip_2)\}\,\langle \mathfrak{S}; (n_1; ip_1; e_1) \mid\mid\mid (n_2; ip_2; e_2)\rangle\,\{\mathsf{True}\}}$$

- Reasoning about network communications (socket protocols)

$$\Phi : \mathit{Message} \to \mathit{iProp} \qquad\qquad a \Mapsto \varPhi$$

  - if $a \Mapsto \varPhi$ and $a \Mapsto \Psi$ then $\varPhi$ and Ψ are equivalent.
  - $a \Mapsto \varPhi \dashv\vdash a \Mapsto \varPhi * a \Mapsto \varPhi$.

Modularity in Aneris logic

- Thread-local reasoning:

$$\frac{\{P_1\}\,\langle n; e_1\rangle\,\{v.Q_1\} \qquad \{P_2\}\,\langle n; e_2\rangle\,\{v.Q_2\}}{\{P_1 * P_2\}\,\langle n; e_1 \mid\mid e_2\rangle\,\{v.\exists v_1, v_2. v = (v_1, v_2) * Q_1[v_1/v] * Q_2[v_2/v]\}}$$

- Node-local reasoning:

$$\frac{\begin{array}{c}\{P_1 * \mathsf{IsNode}(n_1) * \mathsf{FreePorts}(ip_1, \mathfrak{P})\}\,\langle n_1; e_1\rangle\,\{\mathsf{True}\} \\ \{P_2 * \mathsf{IsNode}(n_2) * \mathsf{FreePorts}(ip_2, \mathfrak{P})\}\,\langle n_2; e_2\rangle\,\{\mathsf{True}\}\end{array}}{\{P_1 * P_2 * \mathsf{FreeIp}(ip_1) * \mathsf{FreeIp}(ip_2)\}\,\langle \mathfrak{S}; (n_1; ip_1; e_1) \mid\mid\mid (n_2; ip_2; e_2)\rangle\,\{\mathsf{True}\}}$$

- Reasoning about network communications (socket protocols)

$$\Phi : \mathit{Message} \to \mathit{iProp} \qquad a \Mapsto \Phi$$

  - if $a \Mapsto \Phi$ and $a \Mapsto \Psi$ then Φ and Ψ are equivalent.
  - $a \Mapsto \Phi \dashv\vdash a \Mapsto \Phi * a \Mapsto \Phi$.

Modularity in Aneris logic

- Thread-local reasoning:

$$\frac{\{P_1\}\,\langle n; e_1\rangle\,\{v.Q_1\} \qquad \{P_2\}\,\langle n; e_2\rangle\,\{v.Q_2\}}{\{P_1 * P_2\}\,\langle n; e_1 \mid\mid e_2\rangle\,\{v.\exists v_1, v_2. v = (v_1, v_2) * Q_1[v_1/v] * Q_2[v_2/v]\}}$$

- Node-local reasoning:

$$\frac{\begin{array}{c}\{P_1 * \mathsf{IsNode}(n_1) * \mathsf{FreePorts}(ip_1, \mathfrak{P})\}\,\langle n_1; e_1\rangle\,\{\mathsf{True}\} \\ \{P_2 * \mathsf{IsNode}(n_2) * \mathsf{FreePorts}(ip_2, \mathfrak{P})\}\,\langle n_2; e_2\rangle\,\{\mathsf{True}\}\end{array}}{\{P_1 * P_2 * \mathsf{FreeIp}(ip_1) * \mathsf{FreeIp}(ip_2)\}\,\langle \mathfrak{S}; (n_1; ip_1; e_1) \mid\mid\mid (n_2; ip_2; e_2)\rangle\,\{\mathsf{True}\}}$$

- Reasoning about network communications (socket protocols)

$$\Phi : \mathit{Message} \to \mathit{iProp} \qquad\qquad a \Mapsto \mathit{\Phi}$$

  - if $a \Mapsto \mathit{\Phi}$ and $a \Mapsto \Psi$ then $\mathit{\Phi}$ and Ψ are equivalent.
  - $a \Mapsto \mathit{\Phi} \dashv\vdash a \Mapsto \mathit{\Phi} * a \Mapsto \mathit{\Phi}$.

Specifying socket protocols

$$\Phi : \mathit{Message} \to \mathit{iProp} \qquad a \Mapsto \mathit{\Phi}$$

Given

- a predicate $P : \mathit{String} \to \mathit{iProp}$ for message contents,
- and a predicate $Q : \mathit{Message} \to \mathit{iProp}$

we specify a socket protocol $\mathit{\Phi}$ as follows:

$$\mathit{\Phi}(m) \triangleq \exists \Psi.\, \mathsf{from}(m) \Mapsto \Psi * P(\mathsf{body}(m)) * (\forall m'.\, Q(m') \mathrel{-\!\!*} \Psi(m'))$$

Socket protocols restrict messages and, if necessary, the protocol of the sender!

This is possible because of Iris's impredicativity:

given any $\Phi : \mathit{Message} \to \mathit{iProp}$, $a \Mapsto \mathit{\Phi} : \mathit{iProp}$

Concurrent-Separation-logic-style specifications

Our specifications for distributed systems are inspired by concurrent separation logic

This style of specification lends itself quite well to modular reasoning

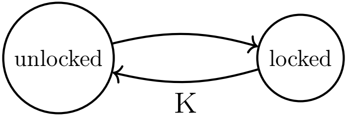
To illustrate this point we see the specs for a distributed lock server

CSL specs for a lock

Lock specifications:

$$
\begin{aligned}
&\exists\, \text{isLock}\,.\\
\wedge\quad & \forall v, K.\ \text{isLock}(v,K) \dashv\vdash \text{isLock}(v,K) * \text{isLock}(v,K)\\
\wedge\quad & \forall v, K.\ \text{isLock}(v,K) \vdash K * K \Rightarrow \mathsf{False}\\
\wedge\quad & \{\mathsf{True}\}\ \text{newLock}\ \ ()\,\{v.\ \exists K.\ \text{isLock}(v,K)\}\\
\wedge\quad & \forall v.\,\{\text{isLock}(v,K)\}\ \text{acquire}\,v\,\{v.K\}\\
\wedge\quad & \forall v.\,\{\text{isLock}(v,K) * K\}\ \text{release}\,v\,\{\mathsf{True}\}
\end{aligned}
$$

Intuitively we think of the following state transition system:



Aneris specs for a distributed lock

$$\Phi_{lock}(m) \triangleq \exists \Psi.\, \mathsf{from}(m) \Mapsto \Psi * (acq(m, \Psi) \vee rel(m, \Psi))$$

$$\begin{aligned}
acq(m, \Psi) \triangleq {} & (\mathsf{body}(m) = \text{"}\texttt{LOCK}\text{"}) * \\
& \forall m'.\, (\mathsf{body}(m') = \text{"}\texttt{NO}\text{"}) \vee (\mathsf{body}(m') = \text{"}\texttt{YES}\text{"} * K) \\
& \qquad -\!\!* \; \Psi(m') \\
rel(m, \Psi) \triangleq {} & (\mathsf{body}(m) = \text{"}\texttt{RELEASE}\text{"}) * K * \\
& \forall m'.\, (\mathsf{body}(m') = \text{"}\texttt{RELEASED}\text{"}) -\!\!* \; \Psi(m')
\end{aligned}$$

Sockets and binding

- Socket creation:

$$\textsc{Socket}$$
$$\{\mathsf{IsNode}(n)\}\ \langle n; \texttt{socket}\ ()\rangle\ \{z.\, z \hookrightarrow_n \mathsf{None}\}$$

- Binding to a static (fixed/primordial) address

$$\textsc{Socketbind-static}$$
$$\{\mathsf{Fixed}(A) * a \in A * \mathsf{FreePort}(a) * z \hookrightarrow_n \mathsf{None}\}$$
$$\langle n; \texttt{socketbind}\ z\ a\rangle$$
$$\{x.\, x = 0 * z \hookrightarrow_n \mathsf{Some}\ a\}$$

- Binding to a dynamic address

$$\textsc{Socketbind-dynamic}$$
$$\{\mathsf{Fixed}(A) * a \notin A * \mathsf{FreePort}(a) * z \hookrightarrow_n \mathsf{None}\}$$
$$\langle n; \texttt{socketbind}\ z\ a\rangle$$
$$\{x.\ x = 0 * z \hookrightarrow_n \mathsf{Some}\ a * a \Mapsto \Phi\}$$

Sending and receiving over sockets

- Sending over a socket

$$\begin{array}{l} \textsc{Sendto} \\ \{z \hookrightarrow_n \mathsf{Some}\ \mathit{from} * \mathit{to} \Mapsto \Phi * \Phi((\mathit{from}, \mathit{to}, \mathit{msg}, \textsc{Sent}))\} \\ \quad \langle n; \texttt{sendto}\ z\ \mathit{msg}\ \mathit{to} \rangle \\ \{x.\ x = |\mathit{msg}| * z \hookrightarrow_n \mathsf{Some}\ \mathit{from}\} \end{array}$$

- Receiving from a socket

$$\begin{array}{l} \textsc{Receivefrom} \\ \{z \hookrightarrow_n \mathsf{Some}\ \mathit{to} * \mathit{to} \Mapsto \Phi\} \\ \quad \langle n; \texttt{receivefrom}\ z \rangle \\ \left\{ \begin{array}{l} x.\ z \hookrightarrow_n \mathsf{Some}\ \mathit{to} * \\ \qquad (x = \texttt{None} \vee (\ \exists m.\, x = \texttt{Some}\,(\mathsf{body}(m), \mathsf{from}(m)) * \Phi(m) * \mathsf{R}(m)\ )) \end{array} \right\} \end{array}$$

Adequacy

Let
– $\mathcal{P} \subseteq Ip$,
– and $A \subseteq Address$.

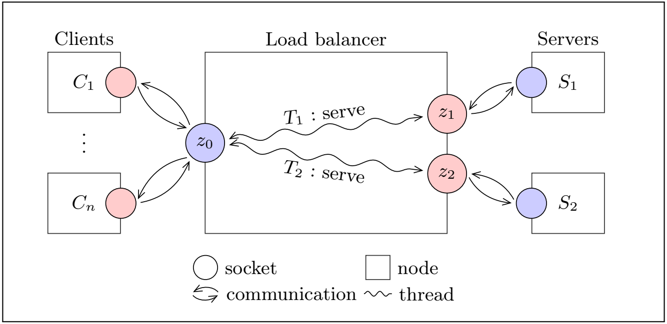
Given a primordial socket protocol Φ_a for each $a \in A$, suppose that the Hoare triple

$$\{\mathsf{Fixed}(A) * \mathop{\LARGE\ast}_{a \in A} a \Mapsto \Phi_a * \mathop{\LARGE\ast}_{i \in \mathcal{P}} \mathsf{FreeIp}(i)\}\ \langle n; e\rangle\ \{v.\mathsf{True}\}$$

is derivable in Aneris.

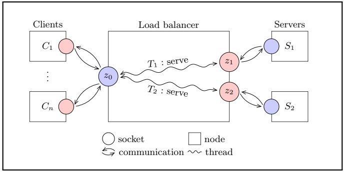
Then, e is safe, *i.e.* it will not crash.

Example: load balancer



$$\Phi_{rel}(P_{val}, P_{in}, P_{out})(m) \triangleq \exists \Psi, v.\, \mathsf{from}(m) \Mapsto \Psi * P_{in}(m, v) * P_{val}(v) * \\ (\forall m'.\, P_{val}(v) * P_{out}(m', v) \mathrel{-\!\!*} \Psi(m'))$$

Example: load balancer



$$\Phi_{rel}(P_{val}, P_{in}, P_{out})(m) \triangleq \exists \Psi, v.\, \mathsf{from}(m) \Mapsto \Psi * P_{in}(m, v) * P_{val}(v) * \\ (\forall m'.\, P_{val}(v) * P_{out}(m', v) \mathrel{-\!\!*} \Psi(m'))$$

$$\left\{\begin{array}{l} \mathsf{Static}((ip, p), A, \phi_{rel}(\lambda\_.\mathsf{True}, P_{in}, P_{out})) * \mathsf{IsNode}(n) * \\ \left(\mathop{\Huge\ast}_{p' \in ports} \mathsf{Dynamic}((ip, p'), A)\right) * \\ \left(\mathop{\Huge\ast}_{s \in srvs} \exists v.\, \mathsf{LB}(1, s, v) * s \Mapsto \phi_{rel}(\lambda v.\, \mathsf{LB}(\tfrac{1}{2}, s, v), P_{in}, P_{out})\right) \end{array}\right\}$$

$$\langle n; \texttt{load\_balancer}\ ip\ p\ srvs \rangle$$

$$\{\mathsf{True}\}$$

More examples

- We use our load balancer specs to verify an addition service:
  - individual servers for adding numbers
  - a load balancing server that distributes the load between worker servers
  - demonstrates horizontal modularity
- We prove correctness of a two phase commit (TPC) library
  - the implementation is parameterized by functions for:
    - voting
    - finalizing (aborting/committing)
  - We use this to implement a replicated logging service:
    - TPC specs guarantee that: either all log servers commit or they all reject the change
    - demonstrates vertical modularity
- We prove correctness of a distributed bag data structure
  - clients can store items in the bag or request items
  - multiple threads on the server respond to requests
  - server-side bag data structure is protected by a lock
  - demonstrates thread-local reasoning

More examples

- We use our load balancer specs to verify an addition service:
  - individual servers for adding numbers
  - a load balancing server that distributes the load between worker servers
  - demonstrates horizontal modularity
- We prove correctness of a two phase commit (TPC) library
  - the implementation is parameterized by functions for:
    - voting
    - finalizing (aborting/committing)
  - We use this to implement a replicated logging service:
    - TPC specs guarantee that: either all log servers commit or they all reject the change
    - demonstrates vertical modularity
- We prove correctness of a distributed bag data structure
  - clients can store items in the bag or request items
  - multiple threads on the server respond to requests
  - server-side bag data structure is protected by a lock
  - demonstrates thread-local reasoning

More examples

- We use our load balancer specs to verify an addition service:
  - individual servers for adding numbers
  - a load balancing server that distributes the load between worker servers
  - demonstrates horizontal modularity
- We prove correctness of a two phase commit (TPC) library
  - the implementation is parameterized by functions for:
    - voting
    - finalizing (aborting/committing)
  - We use this to implement a replicated logging service:
    - TPC specs guarantee that:
      either all log servers commit or they all reject the change
    - demonstrates vertical modularity
- We prove correctness of a distributed bag data structure
  - clients can store items in the bag or request items
  - multiple threads on the server respond to requests
  - server-side bag data structure is protected by a lock
  - demonstrates thread-local reasoning

Thanks!