UNIVERSITÀ
DEGLI STUDI
DI UDINE

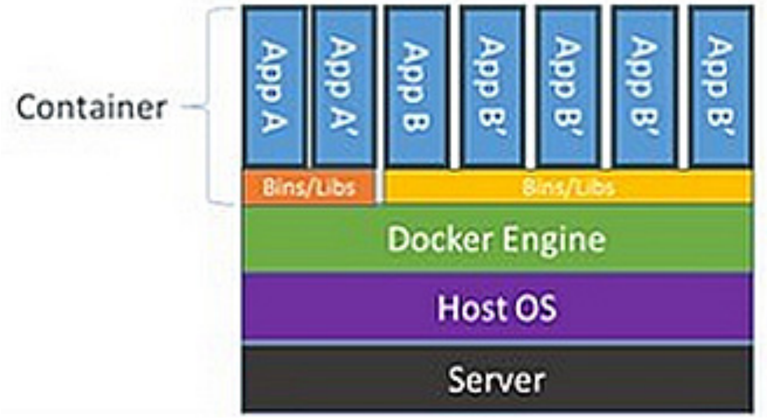
Compositional models for Container-based Systems

Marino Miculan
DMIF, University of Udine / DAIS, Ca’ Foscari University of Venice
marino.miculan@uniud.it

SERICS - SPOKE 6 First Software and Platform Security Workshop
DAIS, Ca’ Foscari University of Venice
October 24, 2023

Microservice-oriented architectures and containers

- **Microservice-oriented architecture**
  - Modern applications are built by composing **microservices** through **interfaces**
  - Distributed, component-based
  - Flexible, scalable, supporting dynamic deployment and reconfiguration, agile programming, etc.
- **Containers** are widely used for implementing Microservices-oriented architectures
  - Lighter than virtual machines
  - Clear definition of interfaces
  - Can be composed

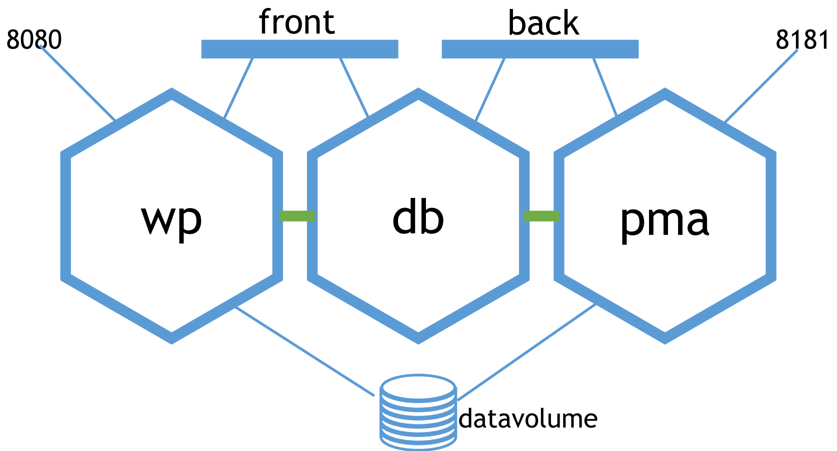


Composition of containers

- Composition is defined by YAML files declaring
  - (Virtual) Networks
  - Volumes (possibly shared)
  - For each container
    - Name
    - Images
    - Networks which are connected to
    - Port remapping for exposed services
    - Volumes
    - Links between services
- Configuration file is fed to a tool (`docker compose`) which downloads the images, creates the containers, the networks, the connections, etc. and launches the system

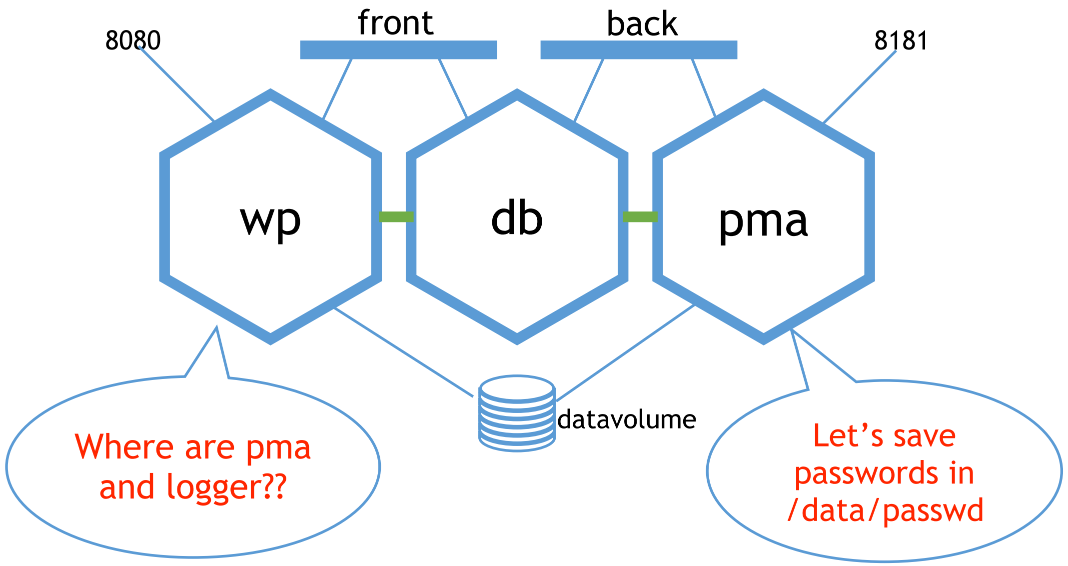
```
services:
  wp:
    image: wordpress
    links:
      - db
    ports:
      - "8080:80"
    networks:
      - front
    volumes:
      - datavolume:/var/www/data:ro
  db:
    image: mariadb
    expose:
      - "3306"
    networks:
      - front
      - back
```

```
  pma:
    image: phpmyadmin/phpmyadmin
    links:
      - db:mysql
    ports:
      - "8181:80"
    volumes:
      - datavolume:/data
    networks:
      - back
networks:
  front:
    driver: bridge
  back:
    driver: bridge
volumes:
  datavolume:
    external: true
```



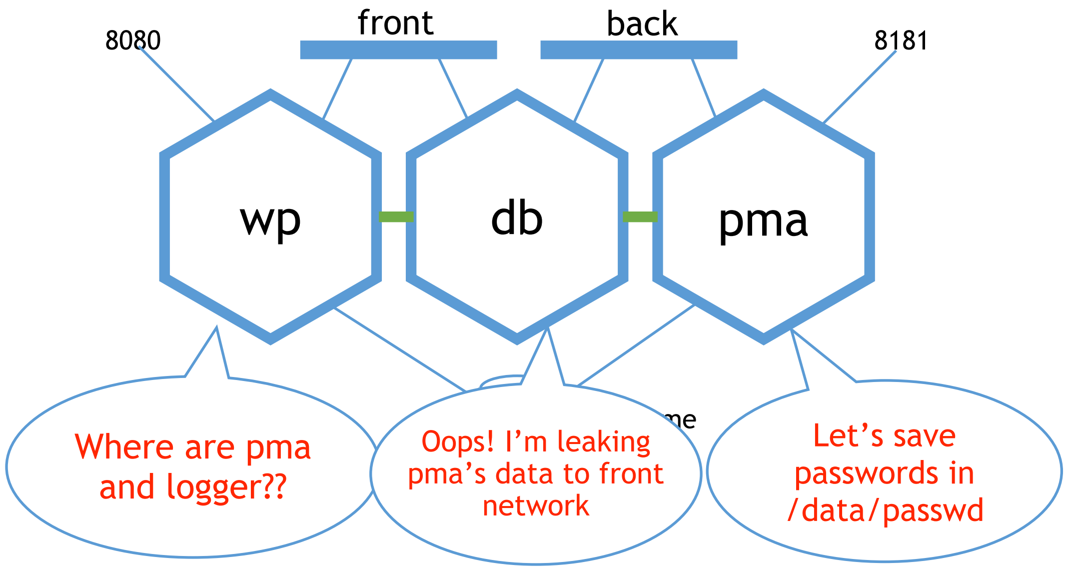
What if a composition configuration is not *correct*?

- A configuration may contain several errors, which may lead to problems during **composition**, or (worse) at **runtime**. E.g.:
  - A container may try to access a **missing services**, or a service which is not connected to by a network
  - **Security policies** violations, e.g. sharing networks or volumes which should not (or only in a controlled way) leading to information leaks



What if a composition configuration is not *correct*?

- A configuration may contain several errors, which may lead to problems during **composition**, or (worse) at **runtime**. E.g.:
    - A container may try to access a **missing services**, or a service which is not connected to by a network
    - **Security policies** violations, e.g. sharing networks or volumes which should not (or only in a controlled way) leading to information leaks
- **Dynamic reconfiguration** can break properties previously valid
    - Container's images can be updated at runtime (e.g. for bug fixing)
    - Adding or removing containers to an existing and running system



Where are pma and logger??

Oops! I'm leaking pma's data to front network

Let's save passwords in /data/passwd

Solid tools need solid theoretical foundations

- We need **tools** for analyzing, verifying (and possibly manipulate) container configurations, before executing the system (static analysis), or at runtime
- We need a ***formal model of containers and services composition***
- Fits SWOPS aims: "**formal techniques** based on secure compilation and **secure composition**, to **reduce the gap between formal models and implementations**"
- This model should support:
  - Composition and nesting of components
  - Dynamic reconfiguration
  - Different granularities of representation
  - Flexibility (can be adapted to various aspects)
  - Openness (we may need to add more details afterwards)
  - …

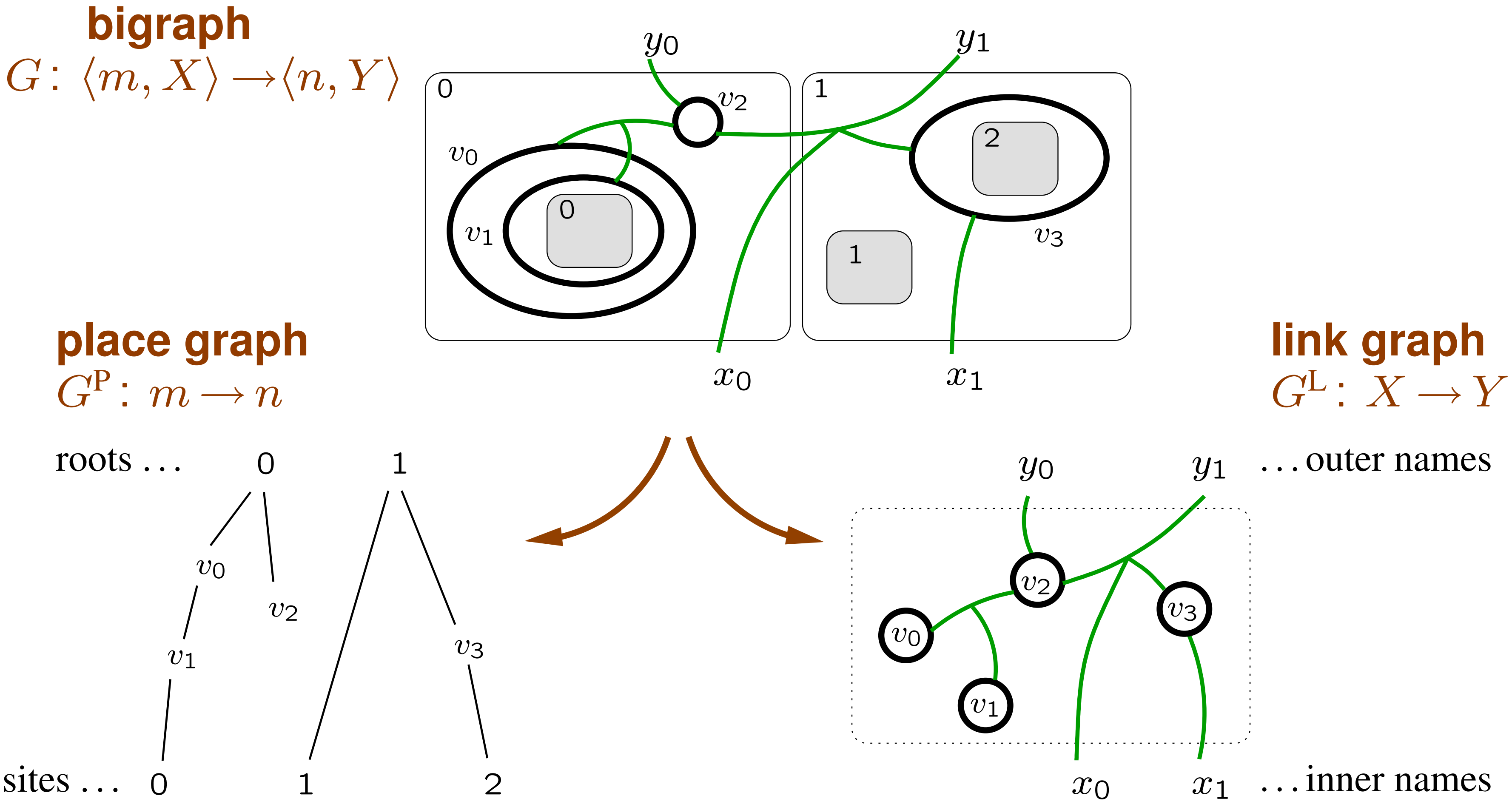
Solid tools need solid theoretical foundations

- We need **tools** for analyzing, verifying (and possibly manipulate) container configurations, before executing the system (static analysis), or at runtime
- We need a ***formal model of containers and services composition***
- Fits SWOPS aims: “**formal techniques** based on secure compilation and **secure composition**, to r plementations”
- This model should
    - Composition ar
    - Dynamic recon
    - Different granularities of representation
    - Flexibility (can be adapted to various aspects)
    - Openness (we may need to add more details afterwards)
    - …

Bigraphs (Milner, 2003): “a general (meta)model for distributed communicating systems, supporting **composition** and **nesting**.”

Quick intro to bigraphs [Milner, 2003]

- A bigraph consists of hyperedges and nodes that can be *nested*. Each hyperedge can connect many ports on different nodes.



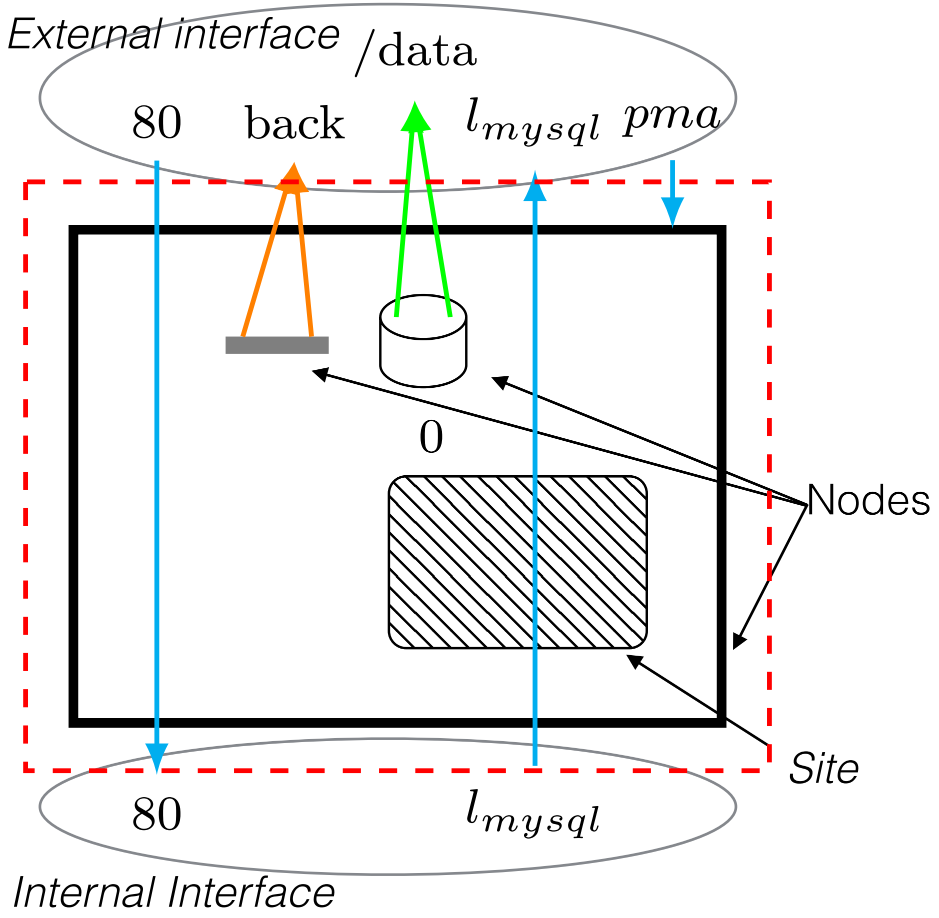
bigraph
$G\colon \langle m, X\rangle \to \langle n, Y\rangle$

place graph
$G^{\mathrm{P}}\colon m \to n$

link graph
$G^{\mathrm{L}}\colon X \to Y$

Local direct bigraphs [Burco, Peressotti, M., ACM SAC 2020]

- For containers, we have introduced **local directed bigraphs**, where
  - Nodes have assigned a type, specifying arity and polarity (represented by different shapes) and can be nested
  - *Sites* represent "holes" which can be filled with other bigraphs
  - Arcs can connect nodes to nodes (respecting polarities) or to names in *internal* and *external interfaces* (with locality)



Local directed bigraphs — more formally

- A *(polarized) interface (with localities)* is a list of pairs of finite sets of names

Global names

Local names (a pair for each locality)

$$X : \langle (X_0^+, X_0^-), (X_1^+, X_1^-), \ldots, (X_n^+, X_n^-) \rangle$$

$$X^+ \triangleq \biguplus_{i=1}^{n} X_i^+ \qquad X^- \triangleq \biguplus_{i=1}^{n} X_i^- \qquad width(X) \triangleq n$$

Ascending names

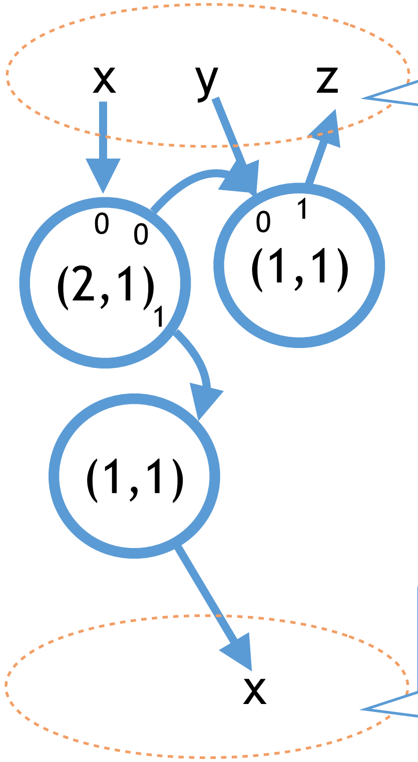
Descending names

- Interfaces can be juxtaposed:

$$X \otimes Y \triangleq \langle (X_0^+ \uplus Y_0^+, X_0^- \uplus Y_0^-), (X_1^+, X_1^-), \ldots, (X_n^+, X_n^-), (Y_1^+, Y_1^-), \ldots, (Y_m^+, Y_m^-) \rangle$$

Local directed bigraphs — more formally

- A **signature** $K = \{c_1, c_2, \dots\}$ is a set of controls, i.e. pairs $c_i = (n_i^+, n_i^-)$
- Each *control* is the type of basic components, specifying inputs (positive part) and outputs (negative part)
- Notice: direction of arrows represents "access" or "usage", not "information flow" (somehow dual to string diagrams)
- Figure aside: a graph representing a system that accesses to some internal service over x, some external service over z, and provides services over x,y



Outer interface

Inner interface

Local directed bigraphs — more formally

- A **signature** $K = \{c_1, c_2, \dots\}$ is a set of controls, i.e. pairs $c_i = (n_i^+, n_i^-)$
- Given two interfaces *I*, *O*, a local directed bigraph $B : I \to O$ is a tuple
$$B = (V, E, ctrl, prnt, link)$$
where
  - V = finite set of *nodes*
  - E = finite set of *edges*
  - $ctrl : V \to K$ = *control map*: assigns each node a type, that is a number of *inward* and *outward ports*
  - *prnt*: tree-like structure between nodes
  - *link*: directed graph connecting nodes' ports and names in interfaces (respecting polarity)

Local directed bigraphs — more formally

- Let K be a fixed signature, and *X*, *Y*, *Z* three interfaces.
- Given two bigraphs $B_1 : X \to Y, B_2 : Y \to Z$, their composition is

$$B_2 \circ B_1 = (V, E, ctrl, prnt, link) : X \to Z$$

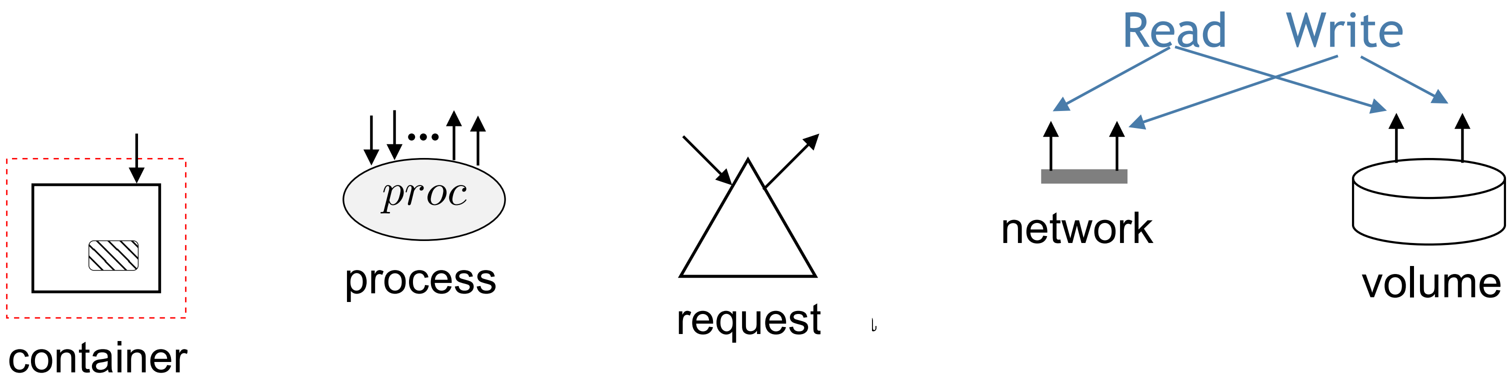
  defined by "filling the holes and connecting the wires" as expected

- Yields a **monoidal category** (Ldb(K),⊗,0)
  - Objects: local directed interfaces
  - Arrows: local directed bigraphs
  - Tensor: juxtaposition
- Enjoys nice properties of bigraphs (RPOs, IPOs, etc.)

A signature for containers

- Controls to represent main elements of a container

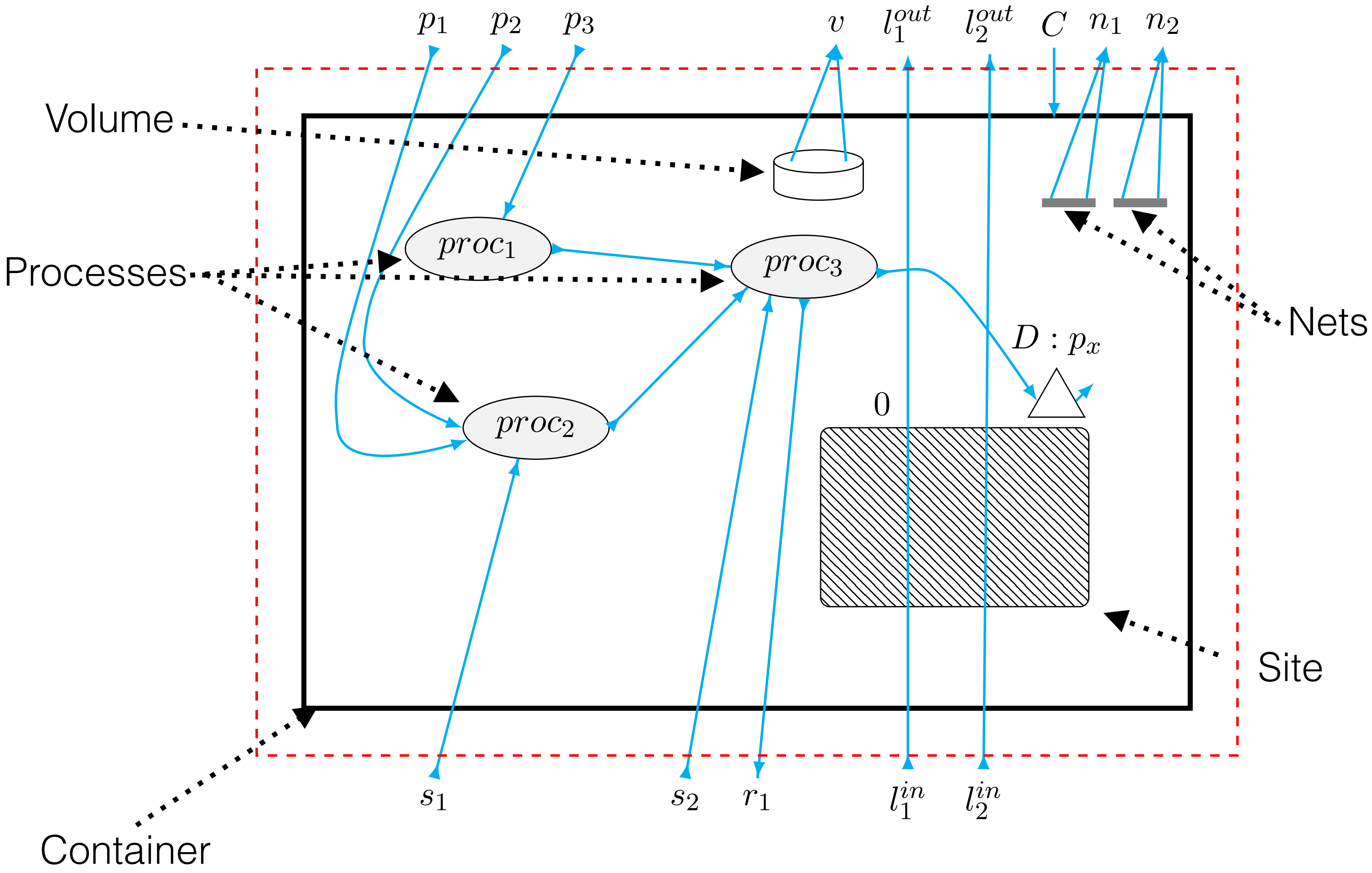
K_{cont} =



- shapes are only for graphical rendering
  - (nodes are subject to some sorting conditions)
- Can be extended with other controls as needed (achieving *flexibility* and *openness*)
  - Changing signature = change of base in fibred category

Containers are local directed bigraphs

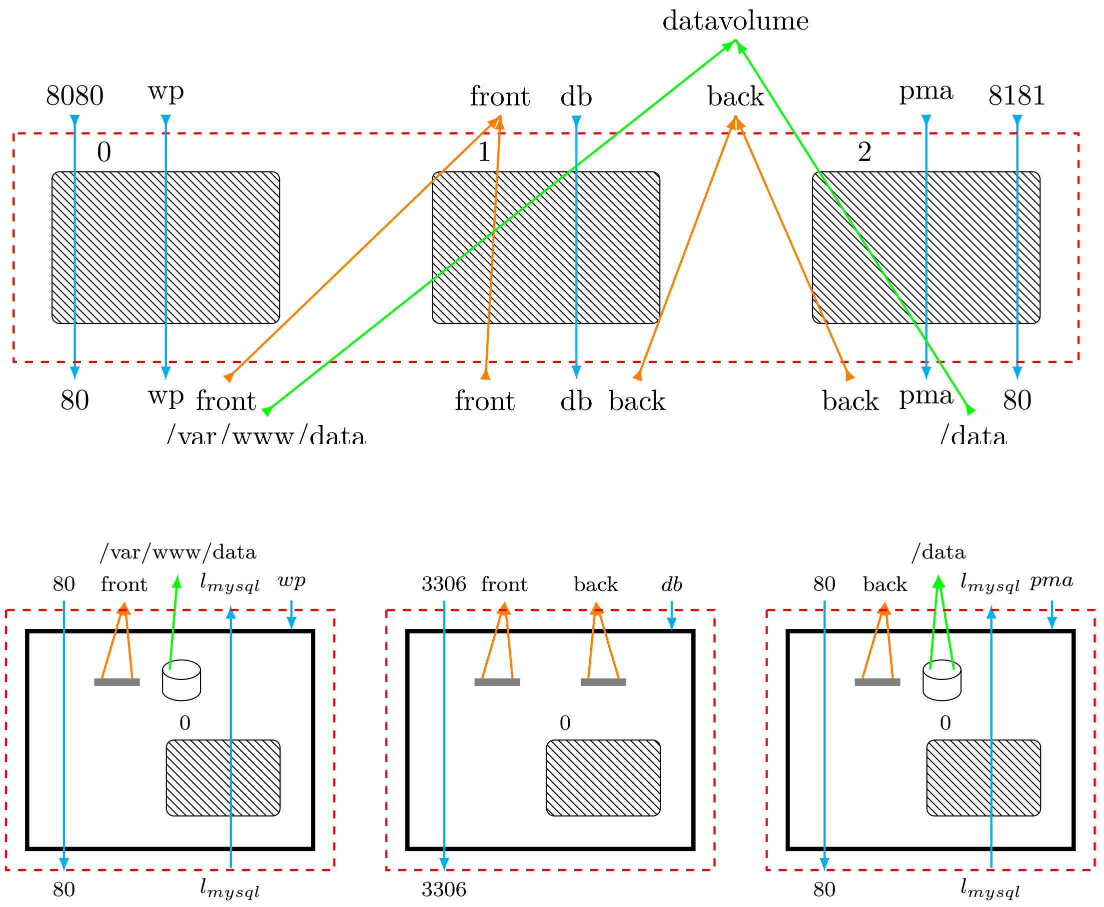
- Container = ldb whose interfaces contain the name of the container, the exposed ports, required volumes and networks, etc.
- This is not only a picture, but the graphical representation of two interfaces and a morphism in the category Ldb(K_{cont})



$$B : \langle(\{\},\{\}),(\{s_1,s_2,l_1^{in},l_2^{in}\},\{r_1\})\rangle \to \langle(\{\},\{\},(\{n_1,n_2,v,l_1^{out},l_2^{out}\},\{p_1,p_2,p_3,C\}))\rangle$$

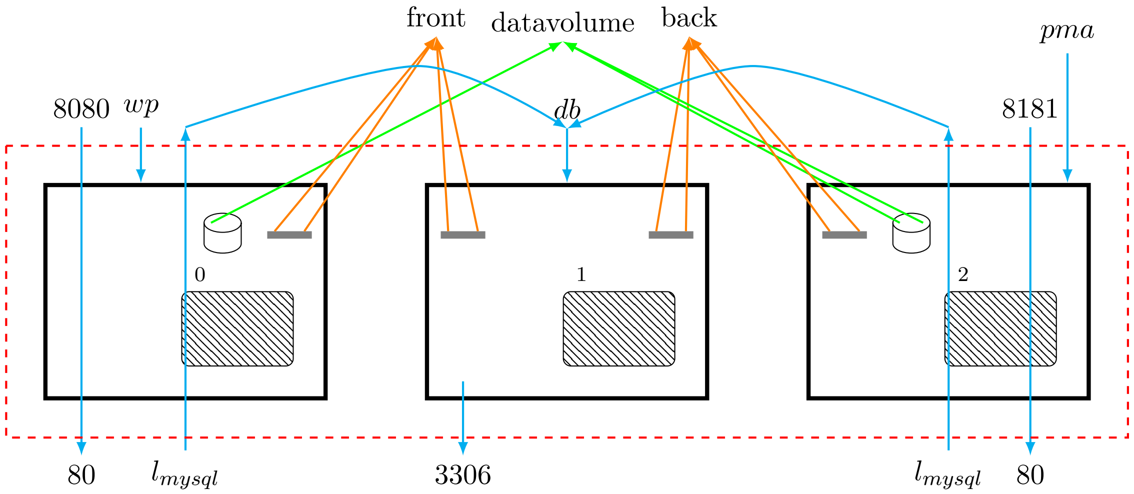
And composition is another bigraph itself

- Composition of containers (as done by `docker-compose`) = composition of corresponding bigraphs inside a *deployment bigraph* specifying volumes, networks, name and port remapping, etc.
  - Encoding is “functorial”



And composition is another bigraph itself

- Composition of containers (as done by `docker-compose`) = composition of corresponding bigraphs inside a *deployment bigraph* specifying volumes, networks, name and port remapping, etc.
  - Encoding is "functorial"
- The deployment bigraph is obtained automatically from the YAML configuration file



Application: safety checks on the configuration

When represented as bigraphs, systems can be analysed using tools and techniques from graph theory

Simple example:

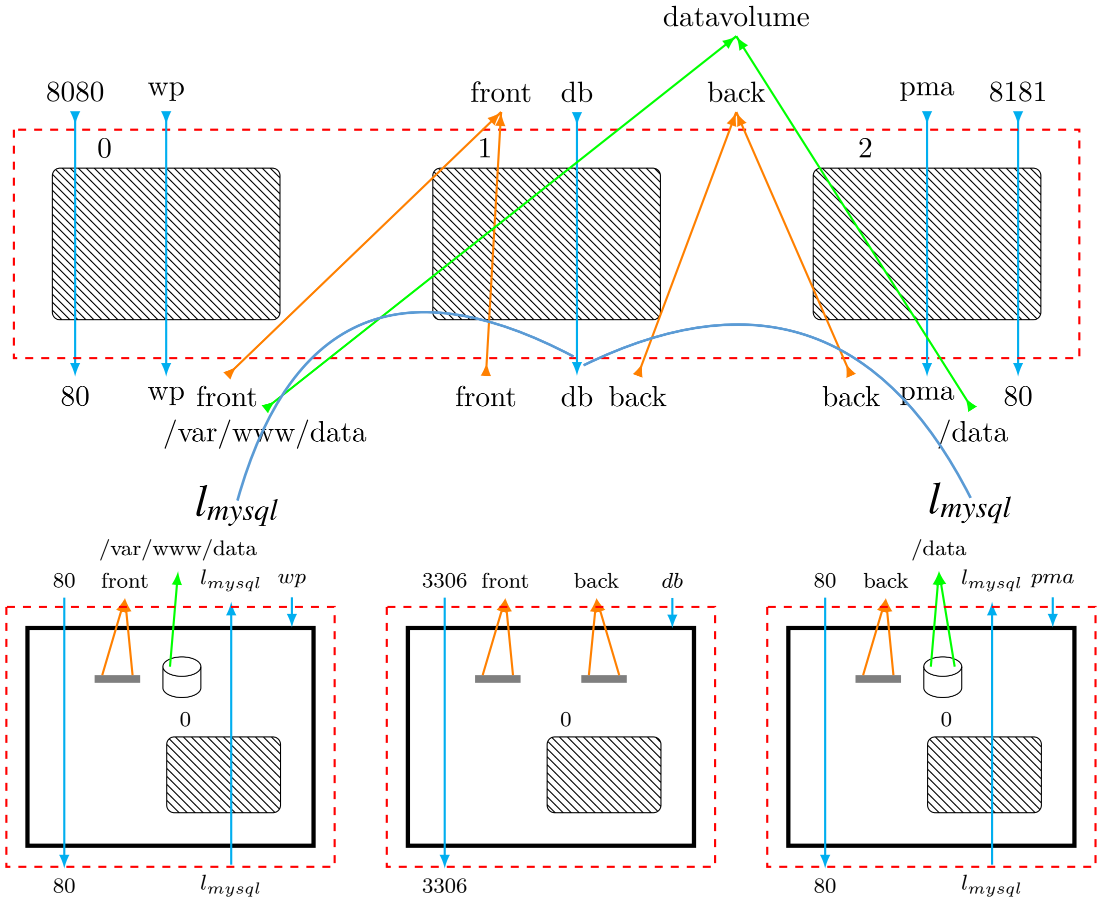
- **Valid links**: “if a container has a link to another one, then the two containers must be connected by at least one network”
  - Corresponds to a simple constraint on the deployment bigraph

Application: safety checks on the configuration

When represented as bigraphs, systems can be analysed using tools and techniques from graph theory

Simple example:

- **Valid links**: “if a container has a link to another one, then the two containers must be connected by at least one network”
  - Corresponds to a simple constraint on the deployment bigraph



Application: Network separation (no information leakage)

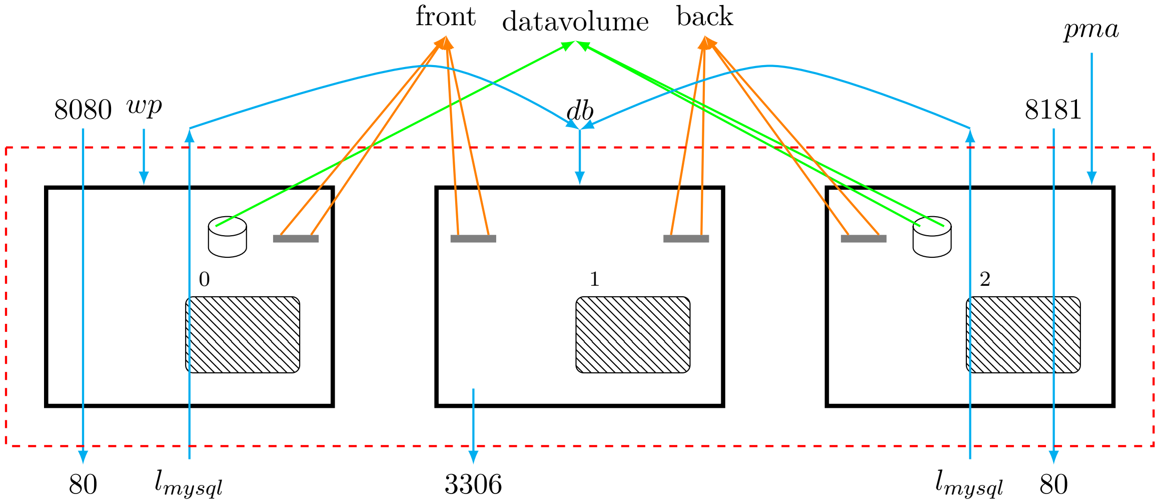
Application: Network separation (no information leakage)

- assume that networks (or volumes) have assigned different security levels (e.g “public < guests < admin”, “back < front”).

Application: Network separation (no information leakage)

- assume that networks (or volumes) have assigned different security levels (e.g “public < guests < admin”, “back < front”).
- Security policy we aim to guarantee:
  - “Information from a higher security network cannot leak into a lower security network, even going through different containers”

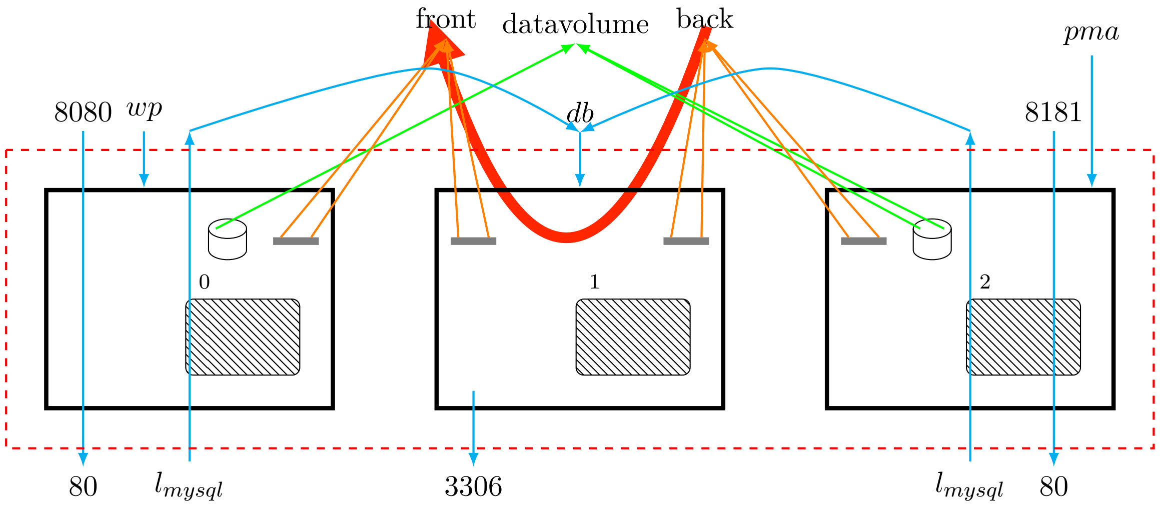
front > back

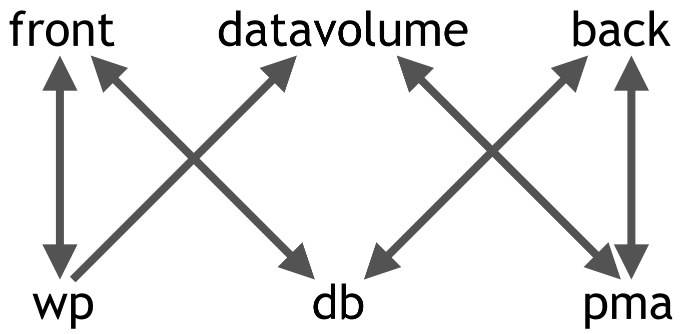


Application: Safe network separation

- Can be reduced to a *reachability problem* on an auxiliary graph representing *read-write accessibility* of containers to resources
  - The r/w accessibility graph is easily derived from the bigraph of the system
- Security policy is reduced to the property: “For each pair of resources m, n such that n < m, there is no directed path from n to m” (i.e., n cannot access m)
  - If this is the case, the configuration respects the security policy. Otherwise, an information leakage is possible

front > back





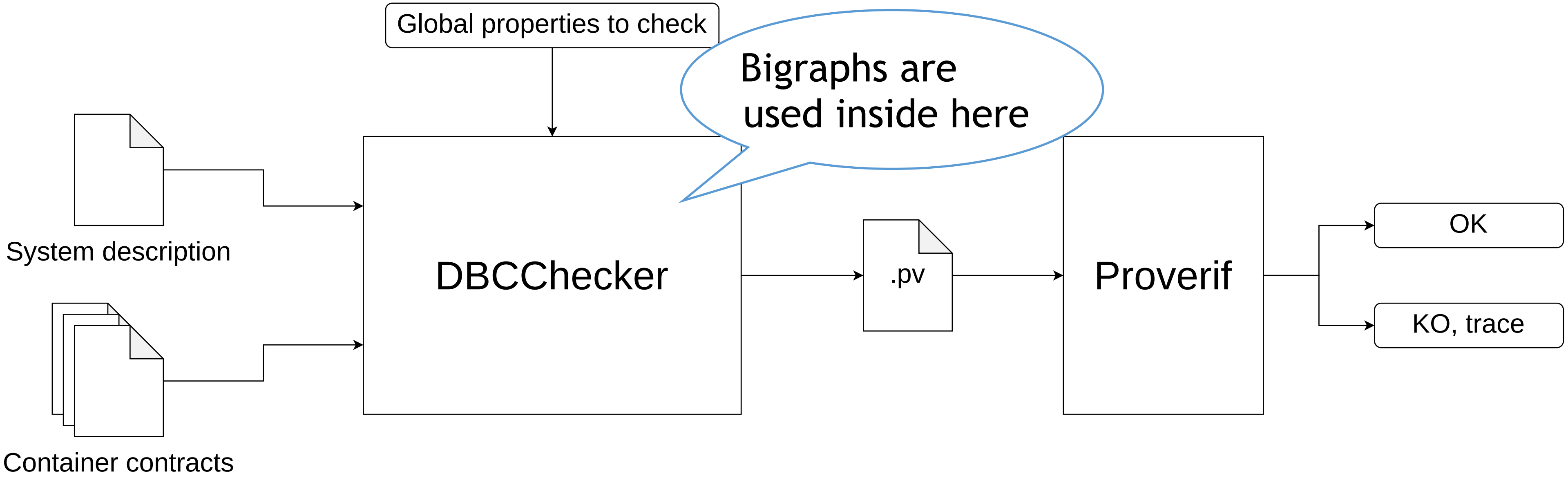
DBCChecker [Altarui, M., Paier, ITASEC 2023]

A tool aiming to verify security properties of systems obtained by composition of containers

DBC CHECKER

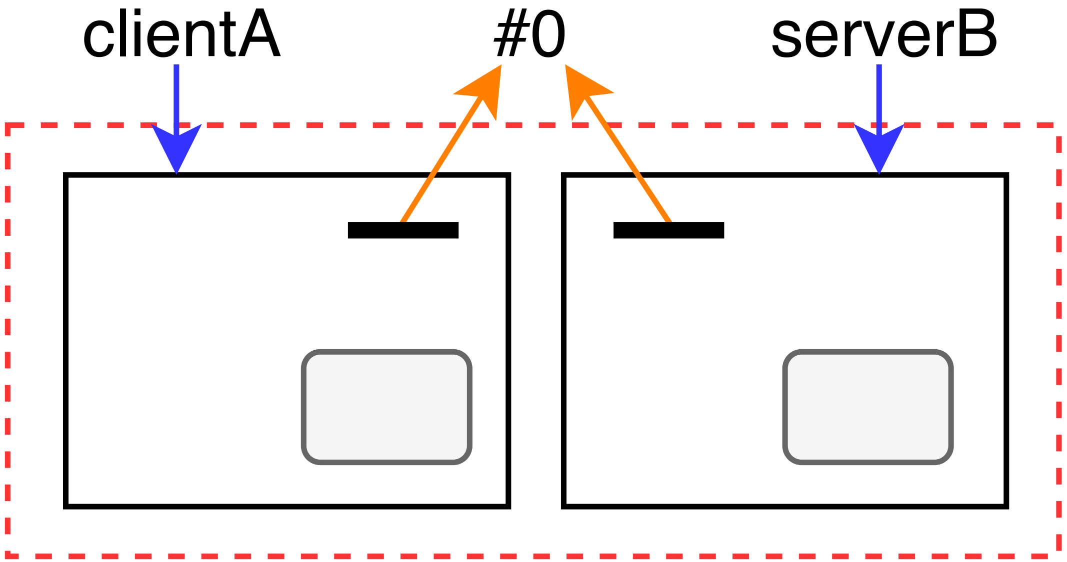
DBCChecker

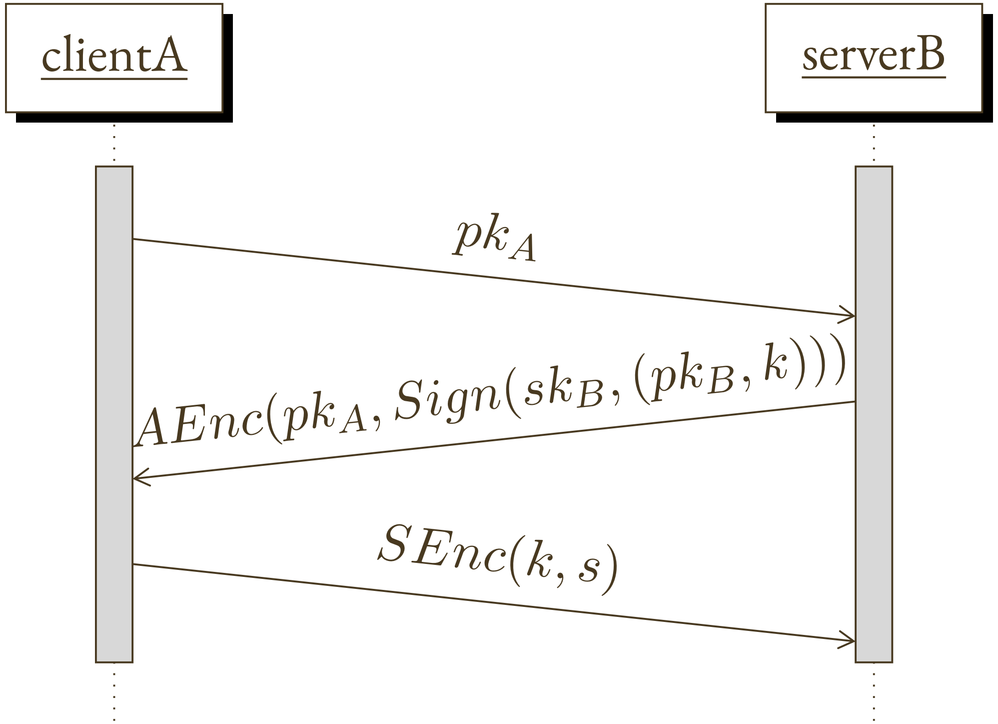
- Input:
  - a configuration of a container-based system (in JBF - JSON Bigraph Format)
  - for each container, an abstract description of the interaction on its interface (“contract”)
  - Global properties to be checked
- Output: a model for the global system, verifiable in some backend



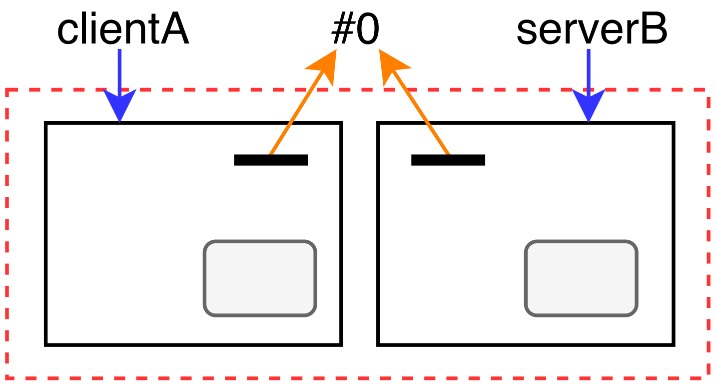
A basic example: secure handshake

- Two containers, “client” and “server”
- Global property to check: confidentiality of message s





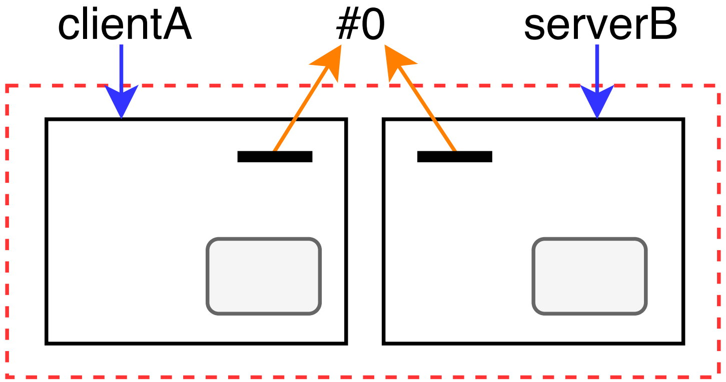
A basic example: secure handshake: contracts

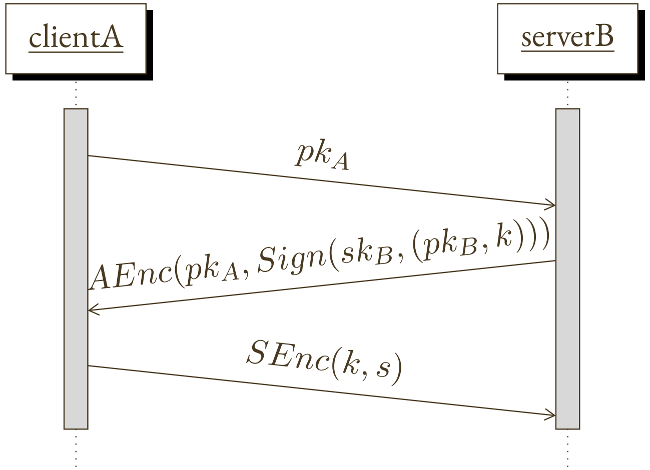


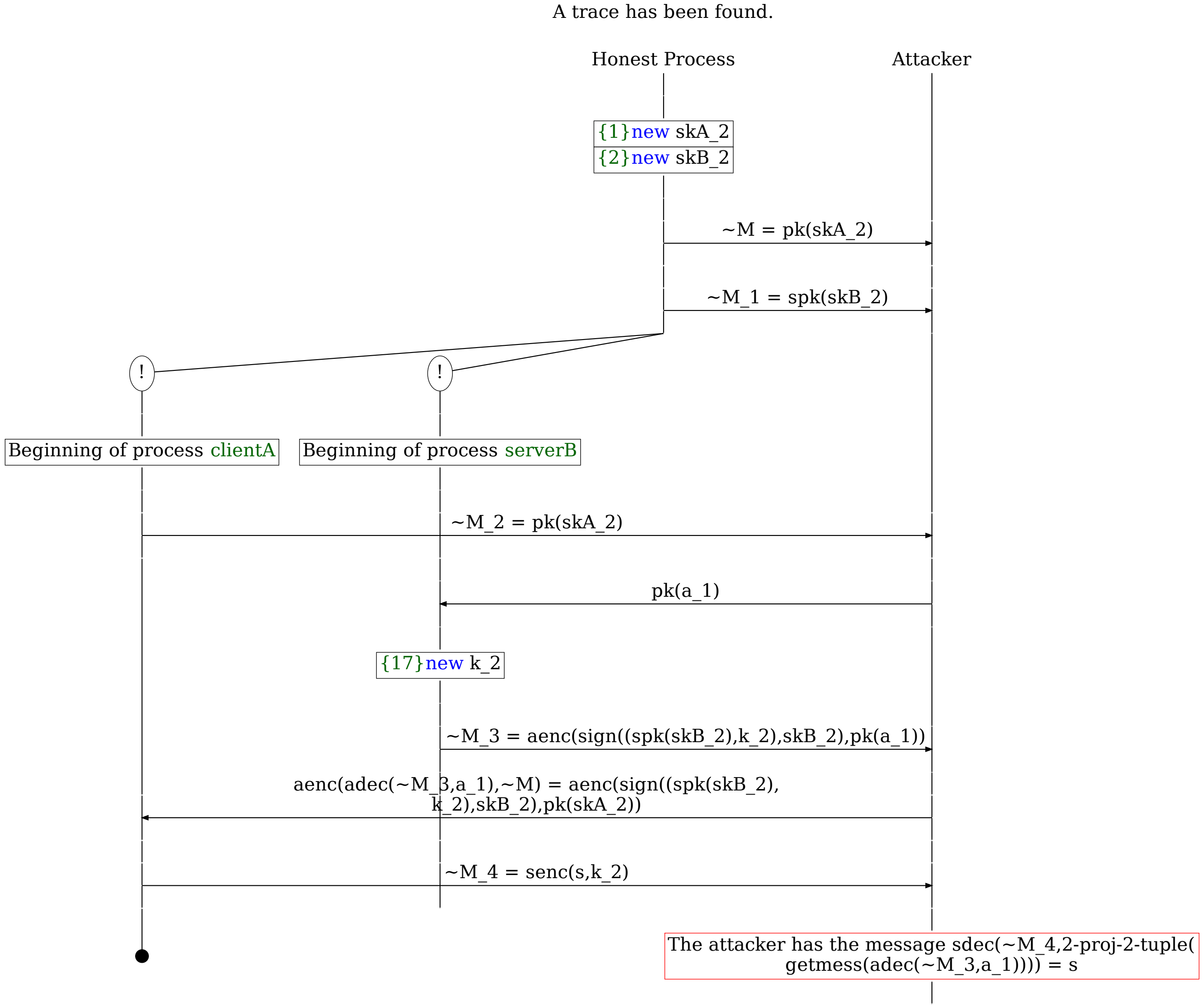
```
    "clientA": {
      "metadata": {
        "type": "node",
        "control": "1on0",
        "params": ["pkA:pkey", "skA:skey",
               "pkB:spkey"],
        "behaviour": "!(out (#0+, pkA);
               in (#0+, x : bitstring);
               let y = adec(x, skA) in
               let (=pkB, k : key) = checksign(y,
               pkB) in
               out (#0+, senc(s, k)))."
        "attribute": ""
      },
      "label": "clientA"
    }
```

```
    "serverB": {
      "metadata": {
        "type": "node",
        "control": "1on0",
        "params": ["pkB:spkey", "skB:sskey"],
        "behaviour": "!(in(#0+, pkX : pkey);
               new k : key;
               out(#0+, aenc(sign((pkB, k), skB),
               pkX));
               in(#0+, x : bitstring);
               let z = sdec(x, k) in 0 ).",
        "attribute": ""
      },
      "label": "serverB"
    }
```

A basic example: secure handshake: analysis result







A trace has been found.

Honest Process

Attacker

{1}new skA_2
{2}new skB_2

~M = pk(skA_2)

~M_1 = spk(skB_2)

!

!

Beginning of process clientA

Beginning of process serverB

~M_2 = pk(skA_2)

pk(a_1)

{17}new k_2

~M_3 = aenc(sign((spk(skB_2),k_2),skB_2),pk(a_1))

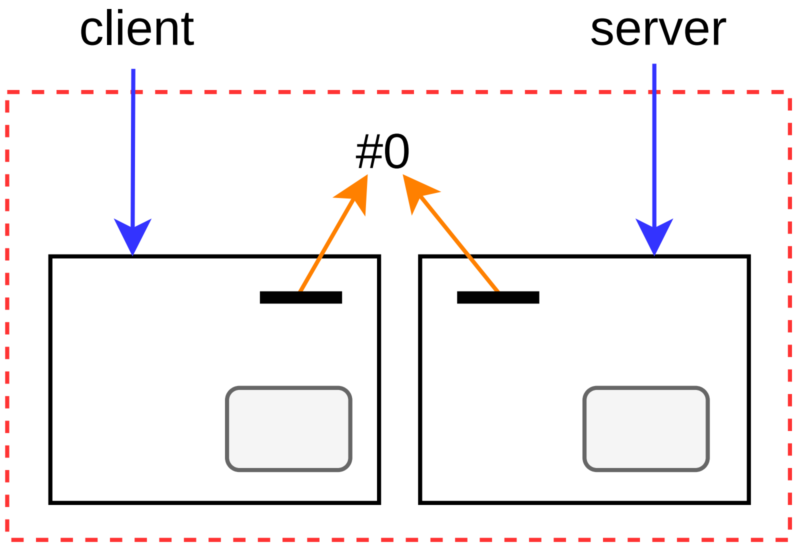
aenc(adec(~M_3,a_1),~M) = aenc(sign((spk(skB_2),
k_2),skB_2),pk(skA_2))

~M_4 = senc(s,k_2)

The attacker has the message sdec(~M_4,2-proj-2-tuple(
getmess(adec(~M_3,a_1)))) = s

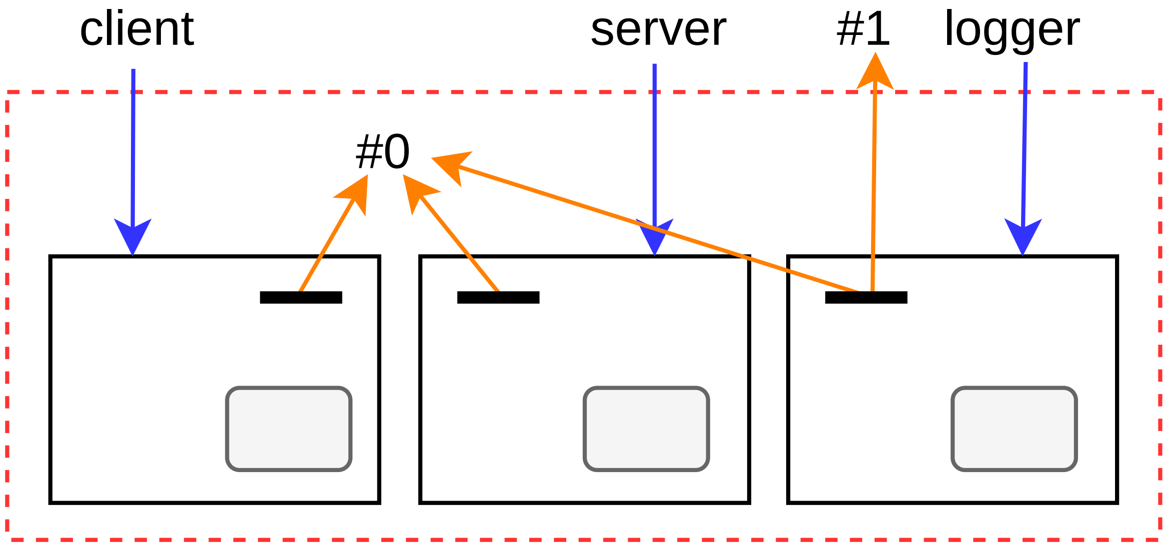
A slightly more advanced example: reconfiguration

- Two containers are communicating over a private channel.
- Global property to check: confidentiality of data.
- The system is secure (because the network is internal).

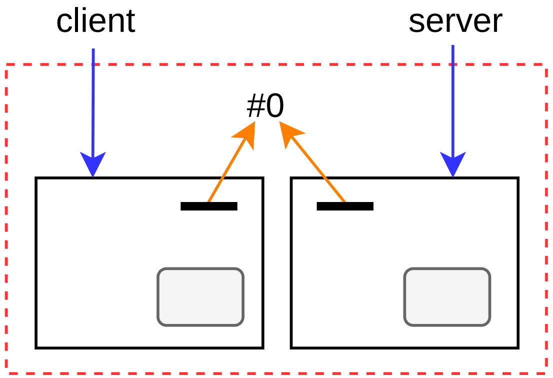


A slightly more advanced example: reconfiguration

- Two containers are communicating over a private channel.
- Global property to check: confidentiality of data.
- The system is secure (because the network is internal).
- But if we add another container, the property may not be preserved



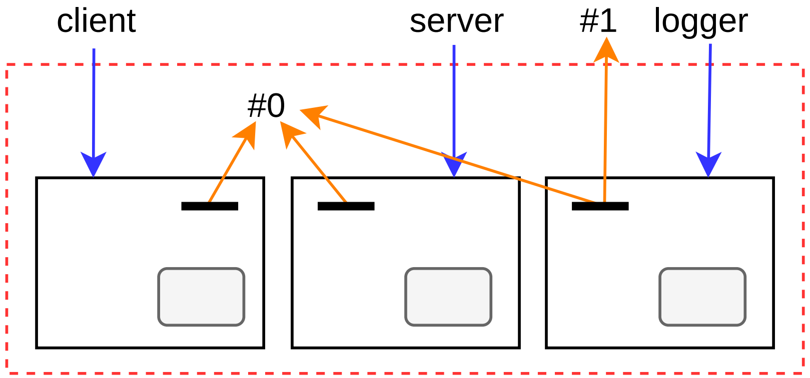
Reconfiguration: contracts



```
"client": {
  "metadata": {
    "type": "node",
    "control": "1on0",
    "properties": {
      "params": [],
      "behaviour": "new
              data:bitstring;
              out(#0-, data).",
      "events": [],
      "attribute": ""
    }
  },
  "label": "client"
},
```

```
"server": {
  "metadata": {
    "type": "node",
    "control": "1on0",
    "properties": {
      "params": [],
      "behaviour": "in(#0-,
             data_received:bitstring).",
      "events": [],
      "attribute": ""
    }
  },
  "label": "server"
},
```

Reconfiguration: contracts



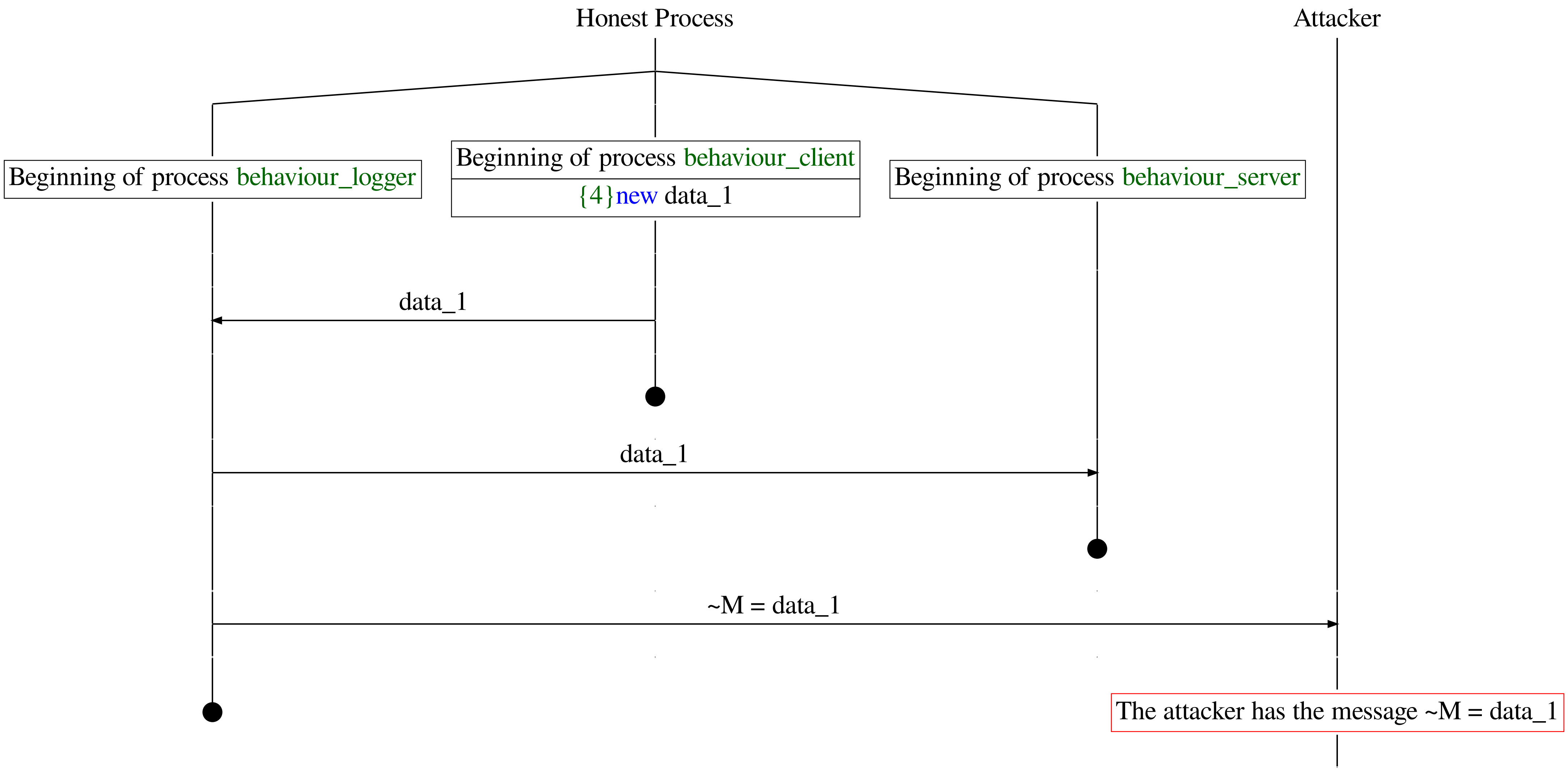
```
"client": {
  "metadata": {
    "type": "node",
    "control": "1on0",
    "properties": {
      "params": [],
      "behaviour": "new
             data:bitstring;
             out(#0-, data).",
      "events": [],
      "attribute": ""
    }
  },
  "label": "client"
},
```

```
"server": {
  "metadata": {
    "type": "node",
    "control": "1on0",
    "properties": {
      "params": [],
      "behaviour": "in(#0-,
             data_received:bitstring).",
      "events": [],
      "attribute": ""
    }
  },
  "label": "server"
},
```

```
"logger": {
  "metadata": {
    "type": "node",
    "control": "2on0",
    "properties": {
      "params": [],
      "behaviour": "in(#0-,
             data_toLog:bitstring)
             out(#0-,
             data_toLog);
             out(#1+,
             data_toLog).",
      "events": [],
      "attribute": ""
    }
  },
  "label": "logger"
},
```

Reconfiguration: analysis result

A trace has been found.



Honest Process

Attacker

Beginning of process behaviour_logger

Beginning of process behaviour_client

{4}new data_1

Beginning of process behaviour_server

data_1

data_1

~M = data_1

The attacker has the message ~M = data_1

Conclusions: some future work

- Formalisation of other static properties (Spatial logics?)
- Finer analysis of containers - i.e., identify connections between processes and resources, by code analysis
- Consider dynamics and temporal properties - in particular, *system reconfiguration*
- Integrate with runtime monitoring
    - If we observe something, which is the new configuration?
- Improve tools, UI/UX
- Quantitative aspects (e.g. fault probability estimation)
- Configuration synthesis
- …

Thanks for your attention! Questions?

marino.miculan@uniud.it