UNIVERSITÀ
DEGLI STUDI
DI UDINE

Bigraphical models for Container-based Systems

Marino Miculan
DMIF, University of Udine
marino.miculan@uniud.it

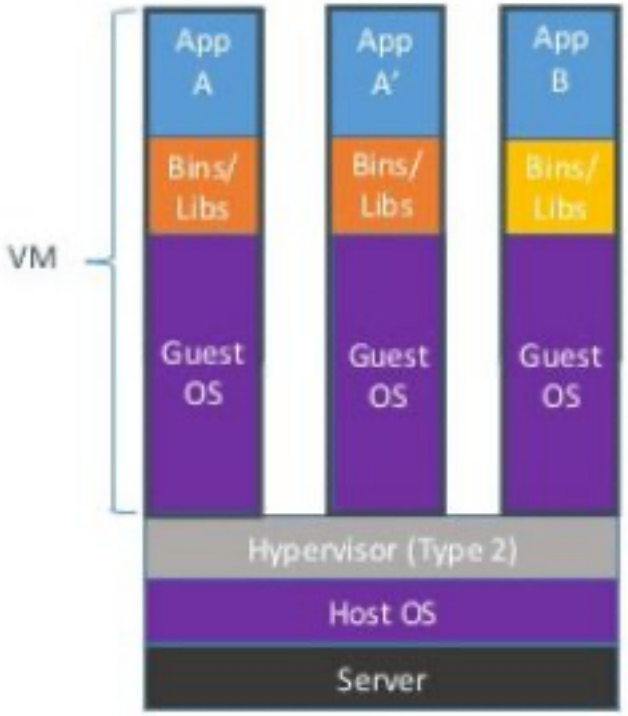
Hot Topics in Language-Based Security Seminar Series,
Department of Computer Science, University of Pisa
May 23, 2023

Microservice-oriented architectures…

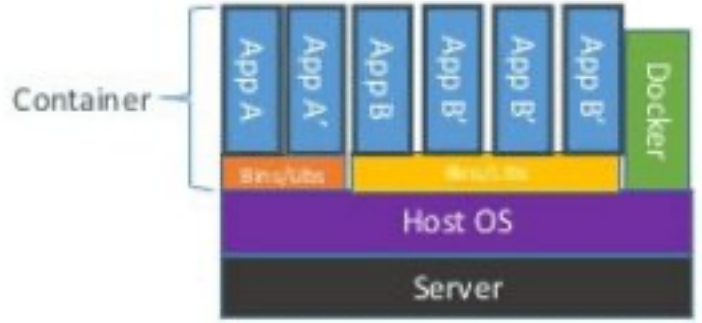
- **Microservice: A highly cohesive, single purpose and decentralized service**
- **Microservice-oriented architecture**
  - Applications are built by composing microservices through **interfaces** (APIs)
  - Distributed component-based
  - Flexible, scalable, supporting dynamic deployment and reconfiguration, agile prog., etc.

… and containers

- **Containers** are widely used for implementing Microservices
  - Ensure execution separation (leveraging kernel namespaces and cgroups in the host OS) separation of tasks, portability
  - **Clear definition of interfaces**
  - Support service and component **composition**
  - Lighter than virtual machines

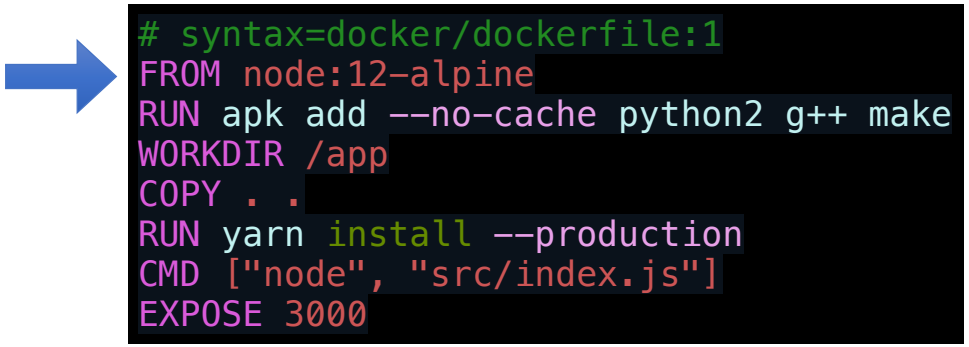


Containers are isolated, but share OS and, where appropriate, bins/libraries

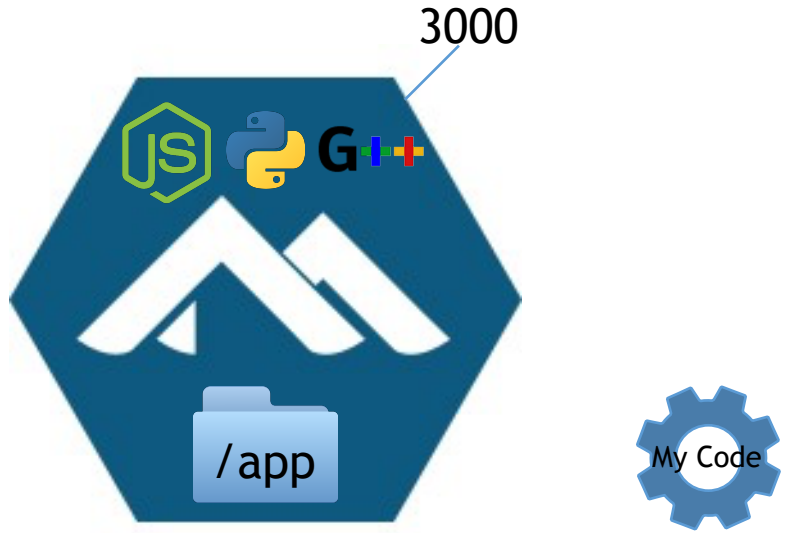


Containers can be filled with libraries, code, data…

- **Dockerfiles**: recipes to build *images*. Example:
  - Start from an existing image
  - Run any command, e.g. to extend the image with any needed package
  - Install programmer's specific code
  - Define the entry point command (what to execute when the container is launched)
  - Declare exposed ports (interfaces)
- These recipes are fed to `docker build`
- Result: **a new image**, which can be run in a container, or used as basis for further builds
- (We will not discuss dockerfiles in this talk)



```
# syntax=docker/dockerfile:1
FROM node:12-alpine
RUN apk add --no-cache python2 g++ make
WORKDIR /app
COPY . .
RUN yarn install --production
CMD ["node", "src/index.js"]
EXPOSE 3000
```

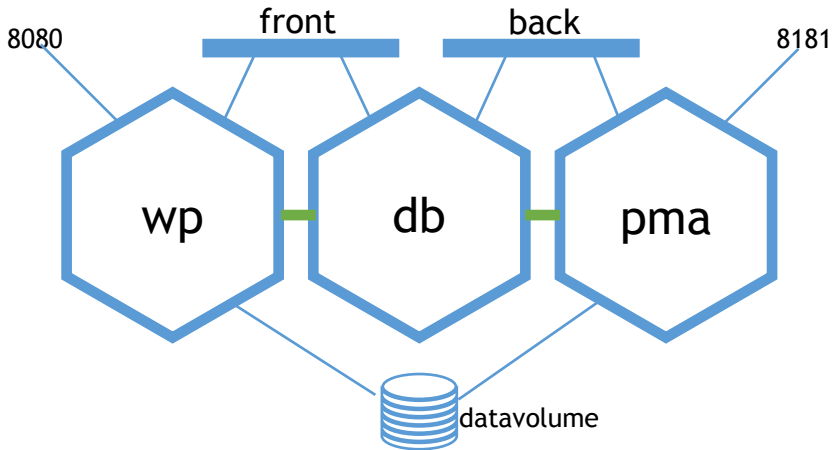


…and connected to other containers through their interfaces

- Composition is defined by YAML files declaring
  - (Virtual) Networks
  - Volumes (possibly shared)
  - For each container
    - Name
    - Images
    - Networks which are connected to
    - Port remappings
    - Volumes
    - Links between services
- Configuration file is fed to an *orchestration* 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
```

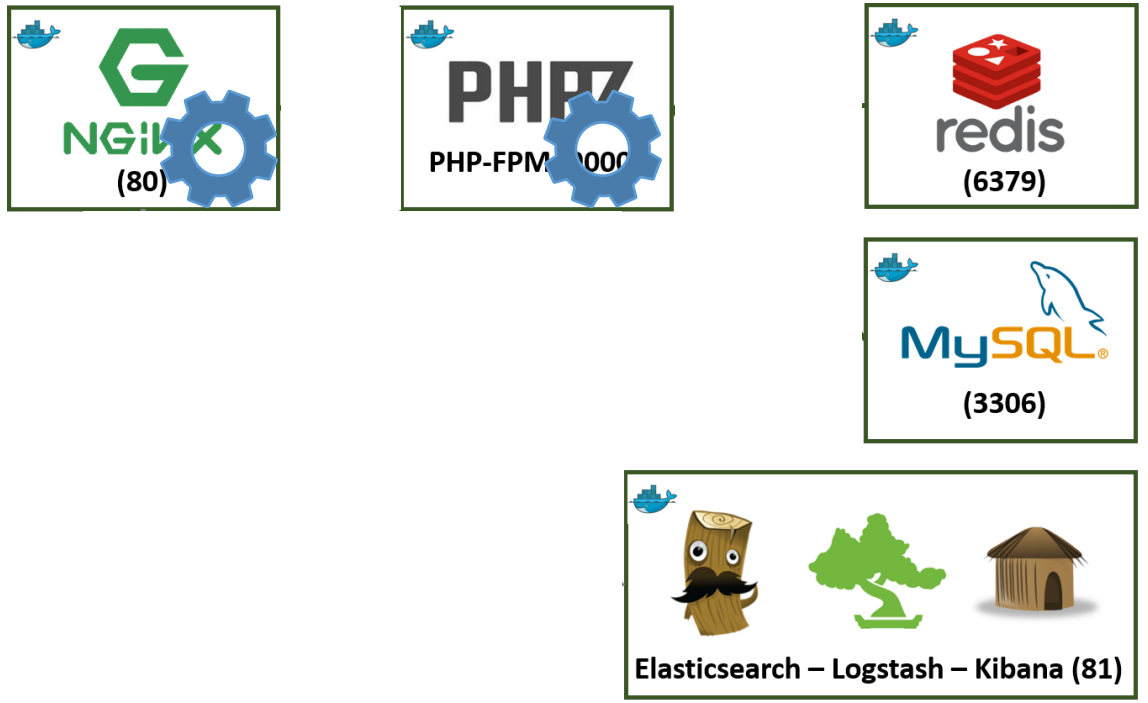
```
  pma:
    image: phpmyadmin/phpmyadmin
    links:
      - db:mysql
    ports:
      - "8181:80"
    volumes:
      - datavolume:/data
    networks:
      - back
networks:
  front:
    driver: bridge
  back:
    driver: bridge
volumes:
  datavolume:
    external: true
```



Vertical vs Horizontal Composition

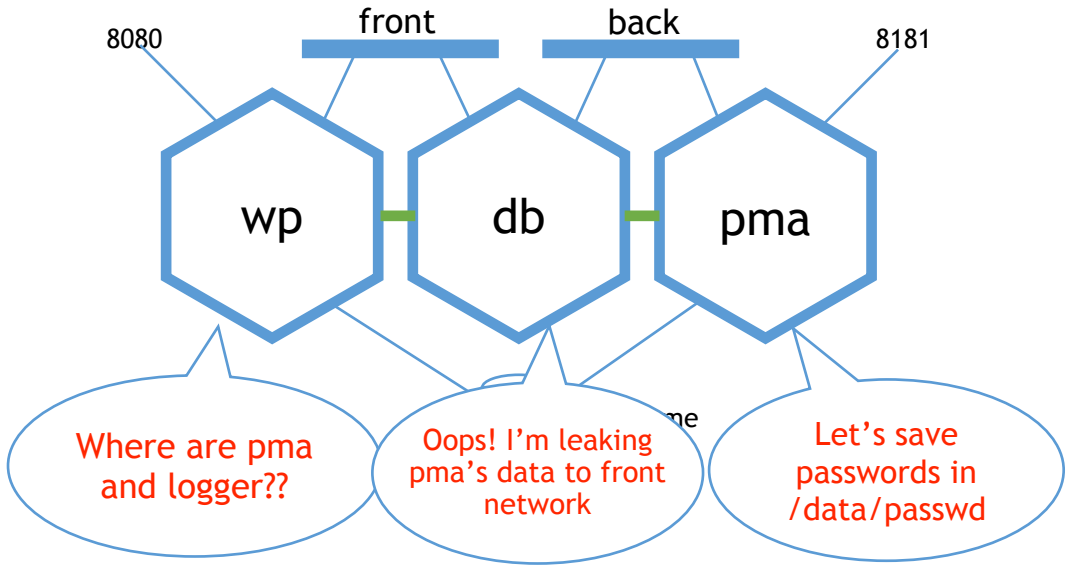
- Containers can be composed to form larger systems
- Two different compositions:
  - **Vertical***: containers can be filled with application specific code, processes… (by developers or at deployment)
  - **Horizontal***: containers are on a par, and communicate through channels (sockets, API), volumes, networks

* = my naming, not official



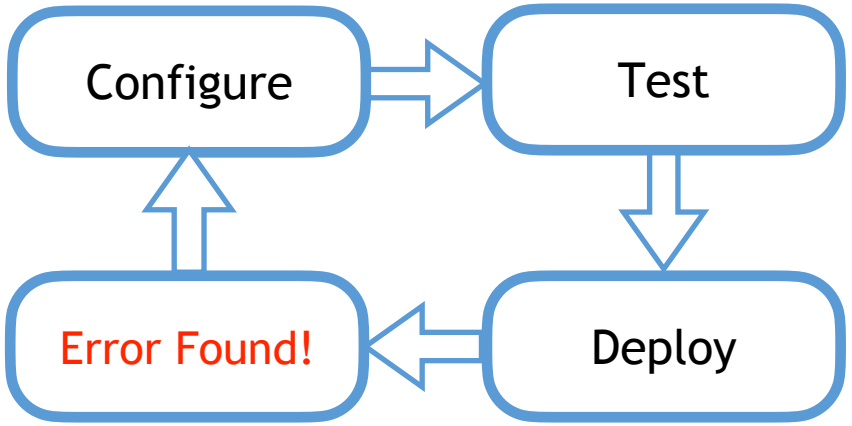
What if a composition configuration is not *correct*?

- A configuration may contain several errors, which may lead to issues 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
  - Ambiguous declaration of services
  - **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



What if a composition configuration is not *correct*?

- Actual composition tools check only very basic aspects
- Common approach: *try-and-error*
  - Expensive
  - Slow
  - Not scalable
  - Not safe enough
- Not acceptable in critical situations
- We need **tools** for analyzing, verifying (and possibly manipulate) container configurations
  - Before executing the system (static analysis), or at runtime



Serious tools need solid theoretical foundations

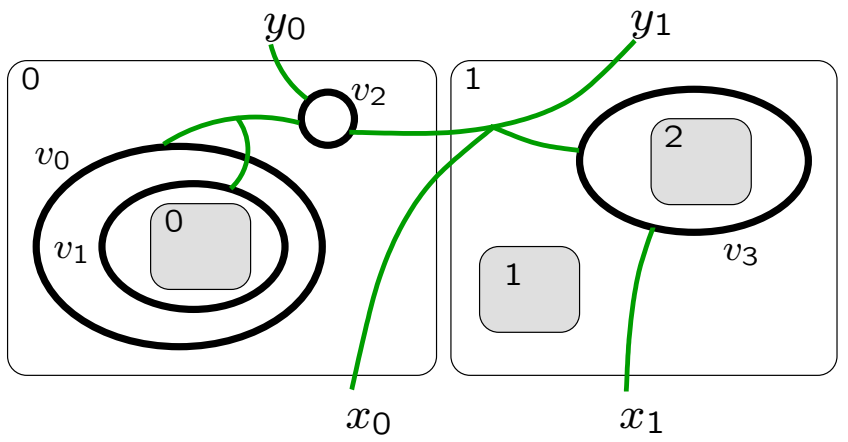
- We need a ***formal model of containers and services composition***
- This model should support:
  - Logical connections of components
  - Horizontal [illegible] omponents
  - Dynamic re[illegible]
  - Different g[illegible]
  - Flexibility
  - Openness (we may need to add more details afterwards)
  - …

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

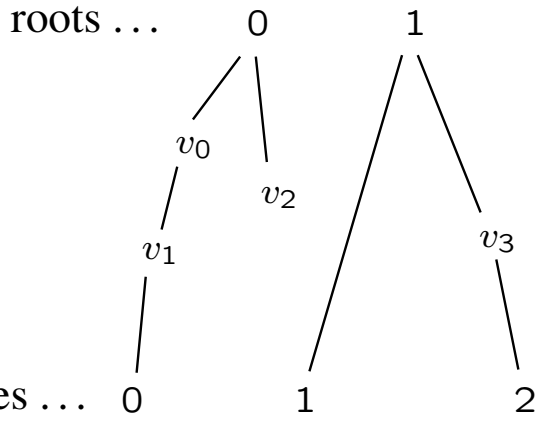
Quick intro to bigraphs [Milner, 2003]

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

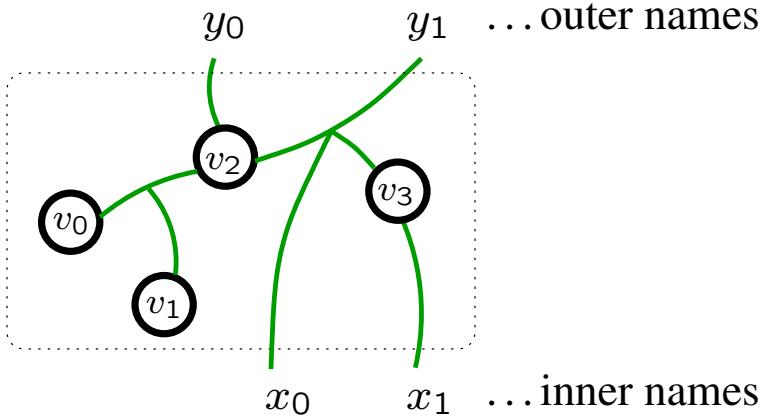
bigraph
$G : \langle m, X \rangle \to \langle n, Y \rangle$



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

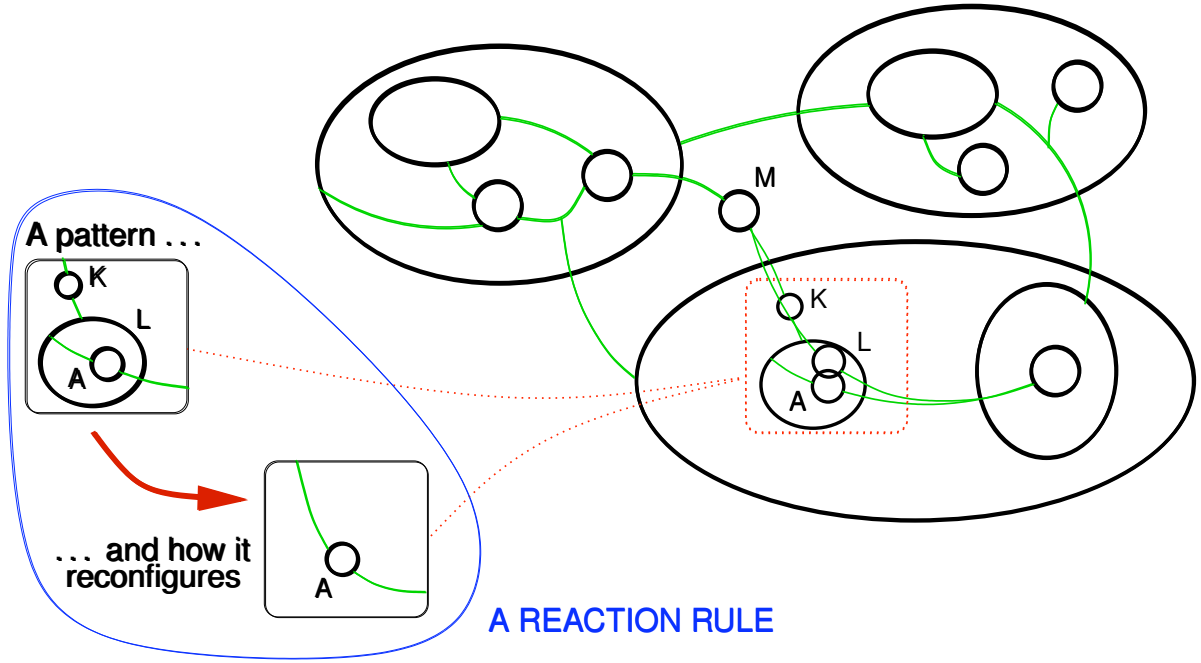


link graph
$G^{\mathrm{L}} : X \to Y$



System modification = Bigraphic rewriting

- State of the system = bigraph
- Dynamics in bigraphs = *graph rewriting rules*
- A rule can replace/move nodes, change connections, etc…



Multi-agent Systems Design and Prototyping with Bigraphical Reactive Systems*

Alessio Mansutti, Marino Miculan, and Marco Peressotti

Bigraphical models for protein and membrane interactions

Giorgio Bacci Davide Grohmann Marino Miculan

A Strategy-Based Formal Approach for Fog Systems Analysis

Souad Marir [1,2,*], Faiza Belala [1] and Nabil Hameurlain [2]

Modeling Self-Adaptive Fog Systems Using Bigraphs

Hamza Sahli[1], Thomas Ledoux[2], and Éric Rutten[3]

Modeling and Verification of Evolving Cyber-Physical Spaces

Christos Tsigkanos, Timo Kehrer, and Carlo Ghezzi
Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Italy

Bigraph Theory for Distributed and Autonomous Cyber-Physical System Design

Vincenzo Di Lecce, Alberto Amato, Alessandro Quarto *Member IAENG,* Marco Minoia

UAV Swarms Behavior Modeling Using Tracking Bigraphical Reactive Systems

Piotr Cybulski * and Zbigniew Zieliński

Controlling resource access in Directed Bigraphs

Davide Grohmann[1], Marino Miculan[2]

BigraphTalk: Verified Design of IoT Applications

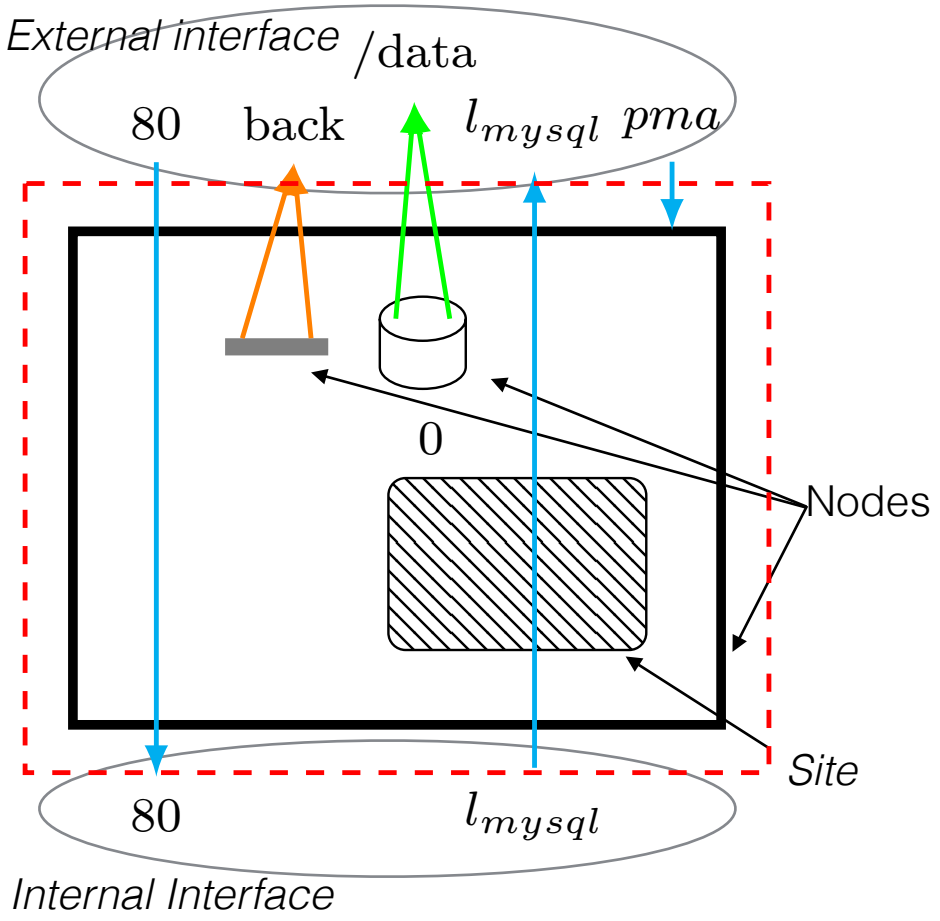
Blair Archibald, Min-Zheng Shieh, Yu-Hsuan Hu, Michele Sevegnani, and Yi-Bing Lin, *Fellow, IEEE*

Security, cryptography and directed bigraphs

Davide Grohmann

Local direct bigraphs [Burco, Peressotti, M., 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:

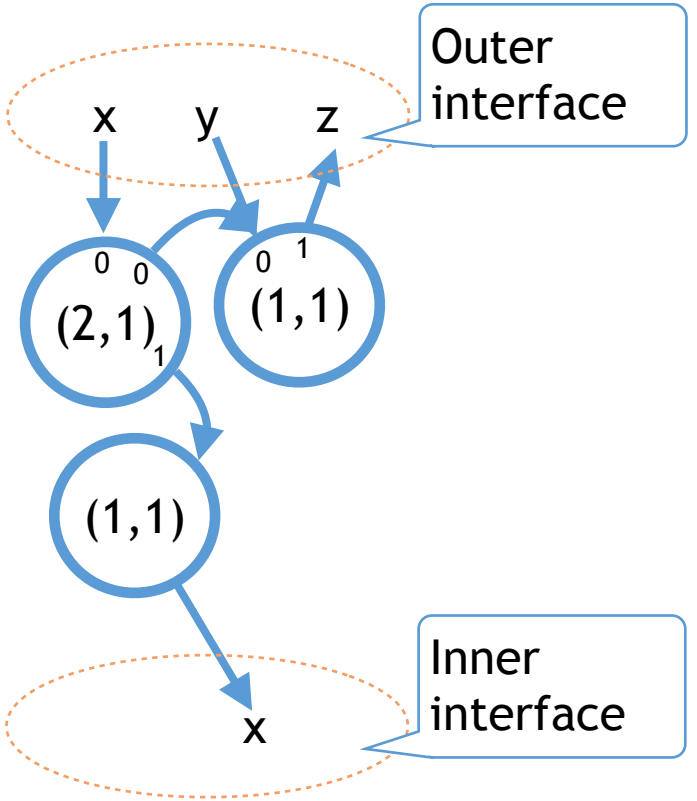
$$X \otimes Y \triangleq \langle (X_0^+ \uplus Y_0^+, X_0^- \uplus Y_0^-), (X_1^+, X_1^-), \ldots, (X_n^+, X_n^-), (Y_1^+, Y_1^-), \ldots, (Y_m^+, Y_m^-) \rangle$$

Local interfaces are everywhere

- This system has an interface (on this side) of width=24
- Each locality (i.e. each socket) has many wires, that is, *names*
  - Ascending names = wires accessing resources outside the PC
  - Descending names = wires giving access to resources inside the PC
- Each locality is for accessing external resources (e.g. energy, mike, network, keyboard, mouse…), or to provide access to internal resources (e.g. PCIe), or both

Local directed bigraphs — more formally

- A **signature** $K = \{c_1, c_2, \ldots\}$ 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)
- **Beware**: direction of arrows represents "access" or "usage" not "information flow" (somehow dual to string diagrams)
- E.g., a graph representing a system that accesses to something internal over x, something external over z, and provides services over x,y



Local directed bigraphs — more formally

- A **signature** $K = \{c_1, c_2, \ldots\}$ 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

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

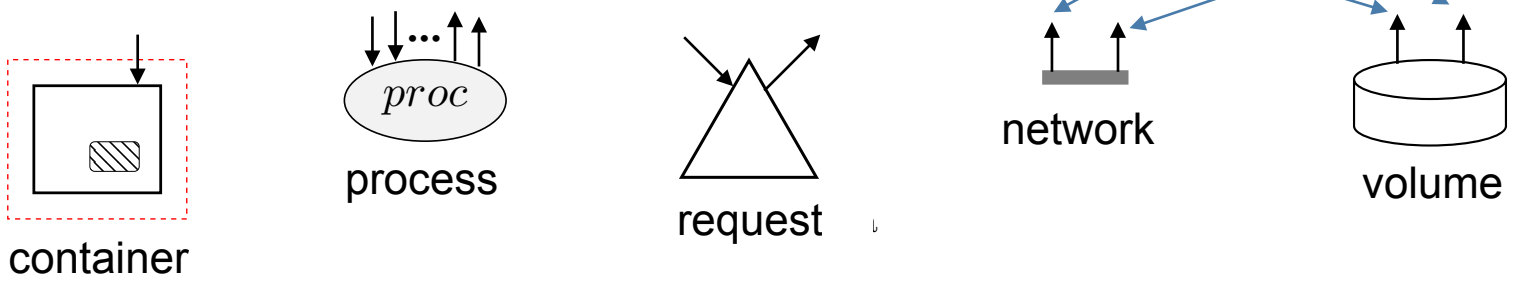
  defined "connecting wires" as expected

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

A LDB signature for containers

- Controls to represent main elements of a container

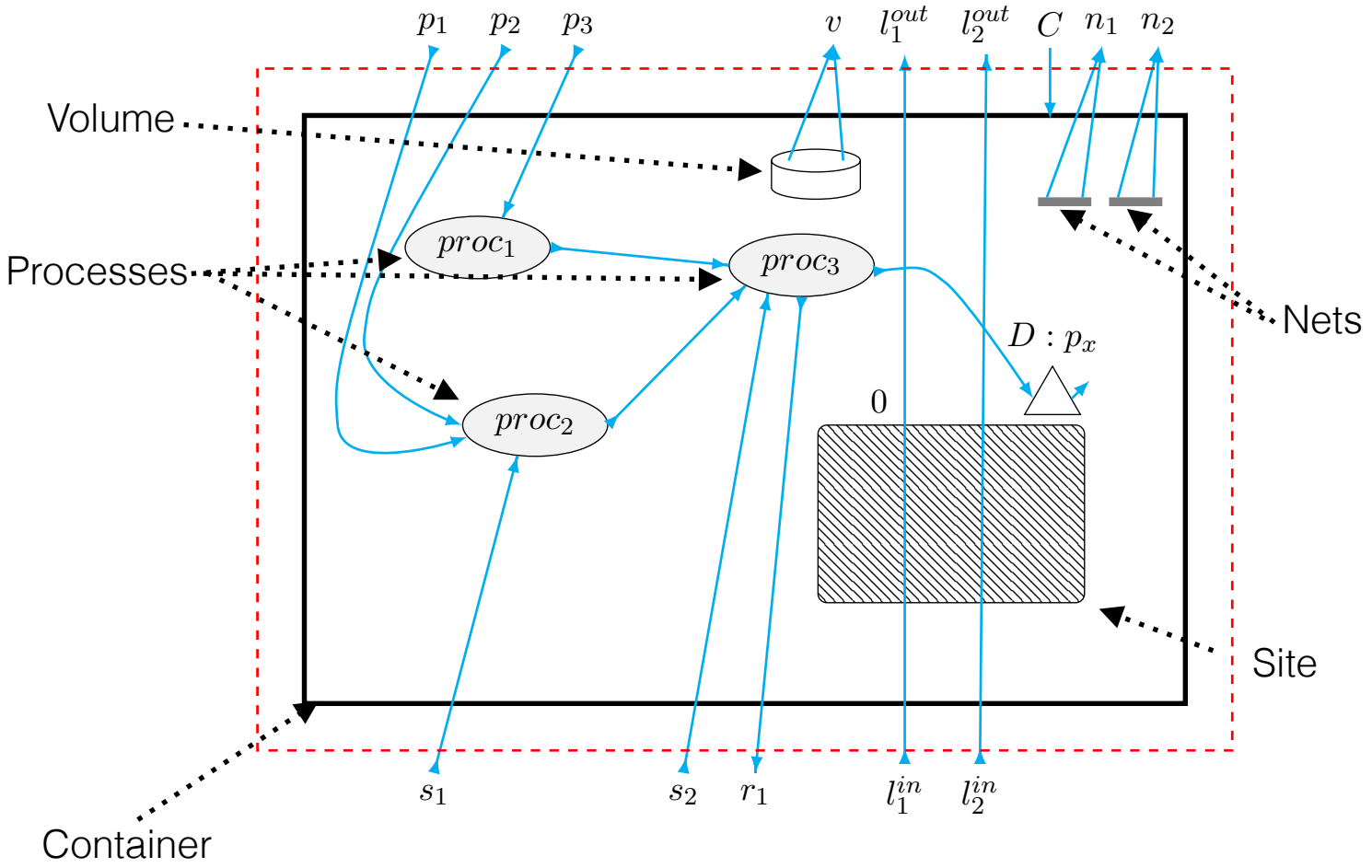
K_{cont} =



- 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!

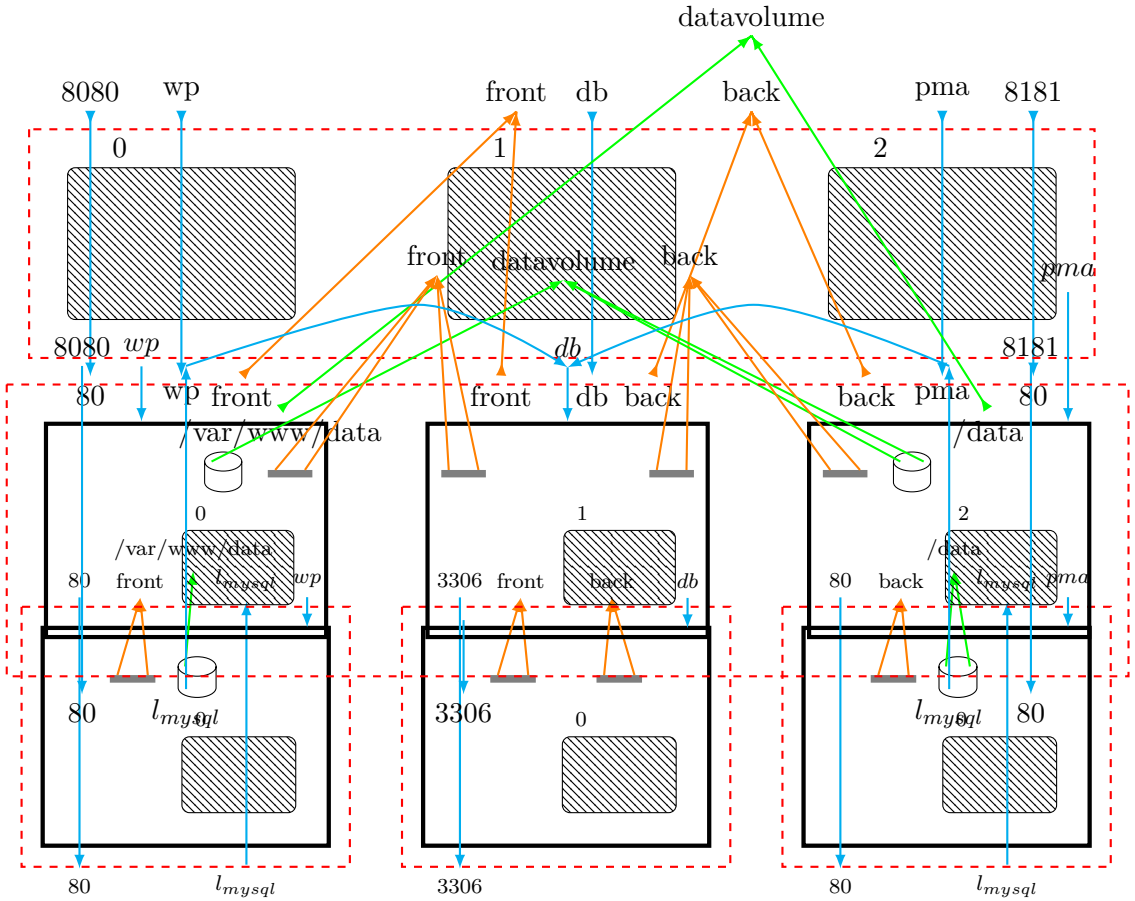
- Container = ldb whose interfaces contain the name of the container, the exposed ports, required volumes and networks, etc.
- This is not only a picture, but the graphical representation of an arrow in the category Ldb(K_{cont})



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

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”
- The deployment bigraph is obtained automatically from the YAML configuration file

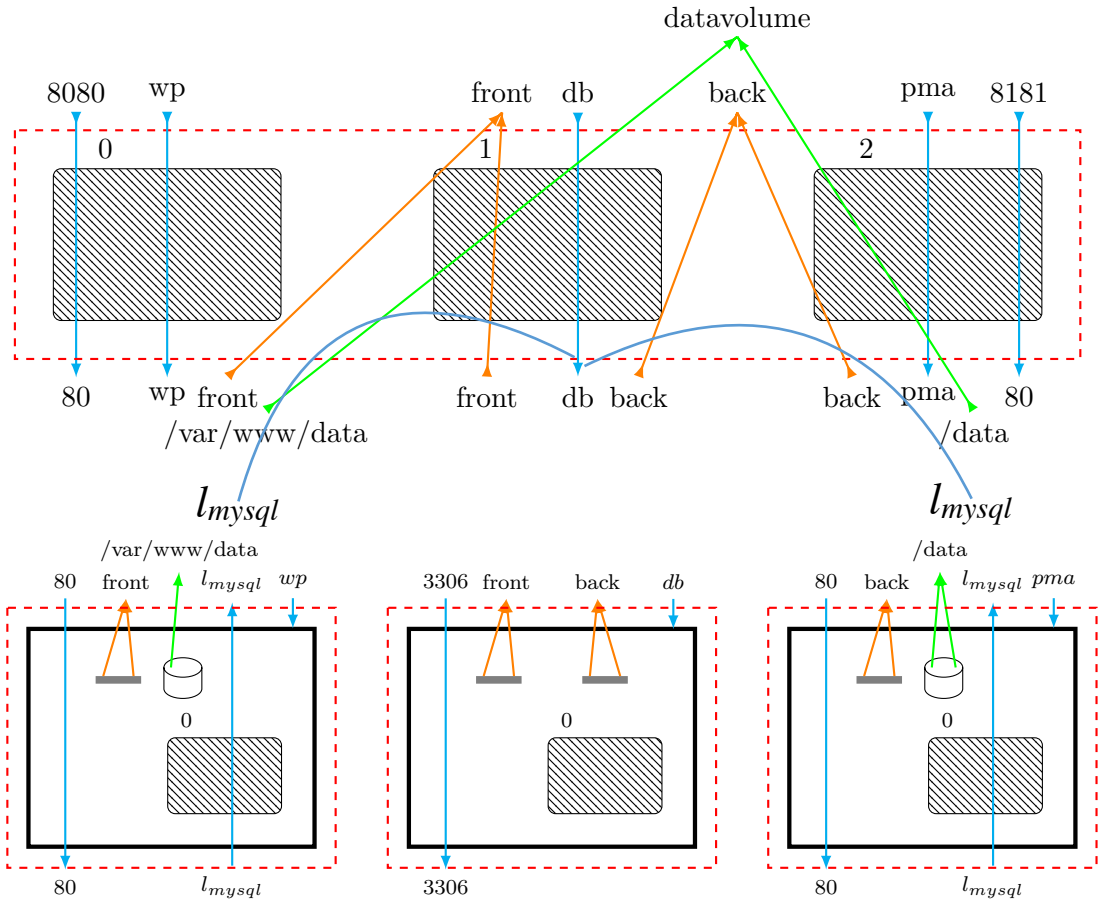


Application: safety checks on the configuration

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

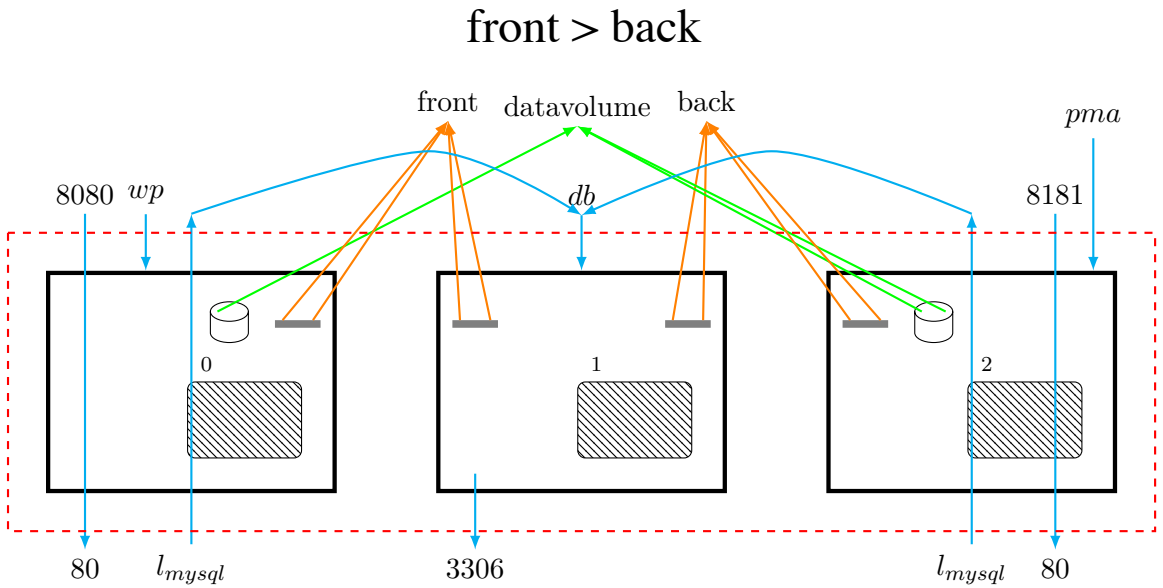
Simple example:

- **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



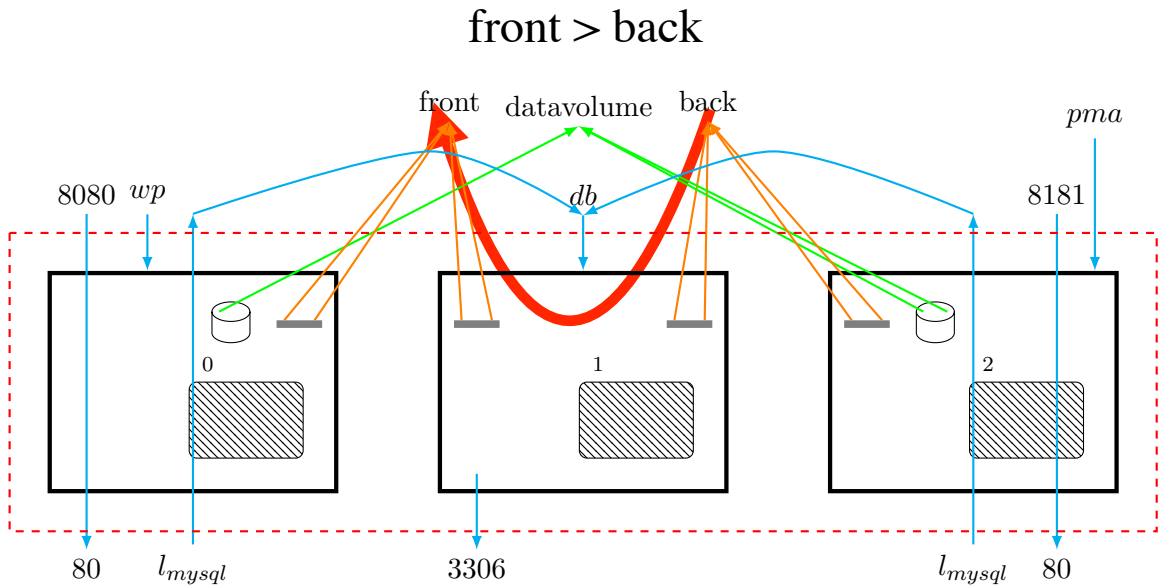
Application: Network separation (no information leakage)

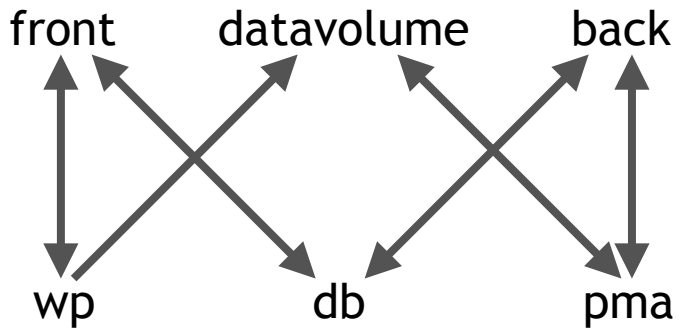
- assume that networks (or volumes) have assigned different security levels (e.g “public < guests < admin”, “back < front”).
- Security policy we aim to guarantee:
  - “Information from a higher security network cannot leak into a lower security network, even going through different containers”



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





Prototype tool: `docker2ldb`

- `docker2ldb` is a CLI tool (written in Java using the *jLibBig* library) which
  - reads a docker-compose configuration file
  - builds the corresponding deployment bigraph object
  - checks for valid connections and network separation

```
01+.back <- {0+@N_55:network, 1+@N_55:network, 0+@N_5A:network, 1+@N_5A:network}
0-@N_52:container <- {pma:i}
0-@N_53:container <- {db:i, l_db_wp:i, l_mysql_pma:i}
0-@N_54:container <- {wp:i}
0 <- {N_52:container, N_53:container, N_54:container}
N_52:container <- {N_55:network, N_57:volume, 0}
N_53:container <- {N_58:network, N_5A:network, 1}
N_54:container <- {N_5D:network, N_5B:volume, 2}
N_55:network <- {}
N_57:volume <- {}
N_58:network <- {}
N_5A:network <- {}
N_5D:network <- {}
N_5B:volume <- {}
[WARNING] Network "back" can read network "front"!
```

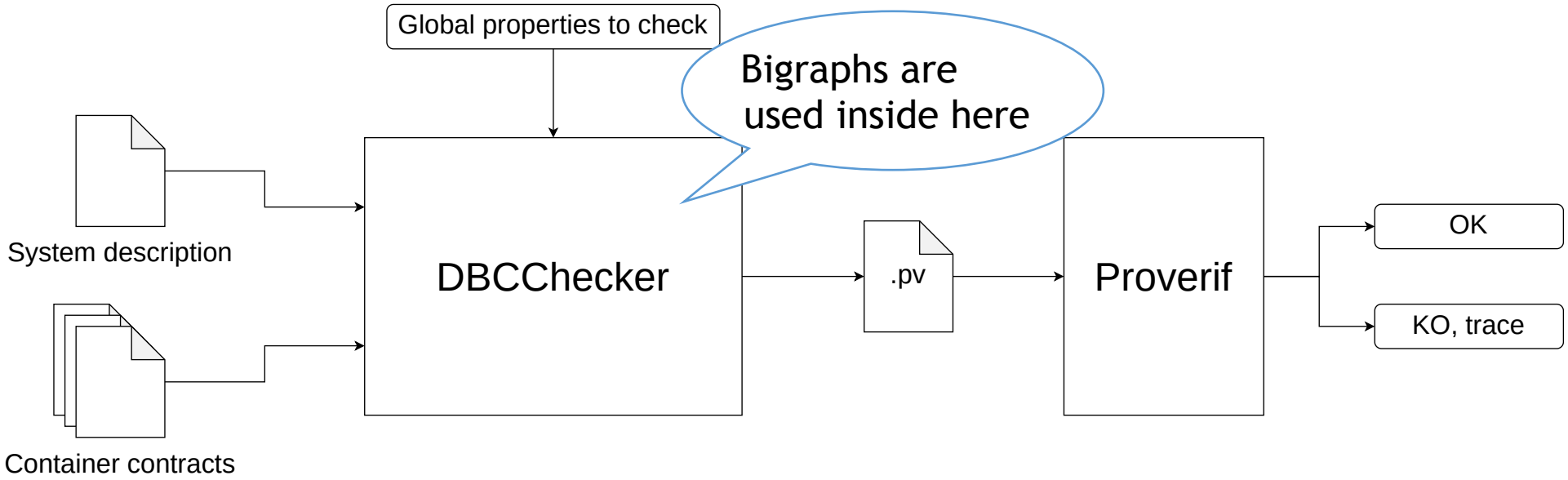
DBCChecker [Altarui, M., Paier, 2023]

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

D B C
C H E C K E R

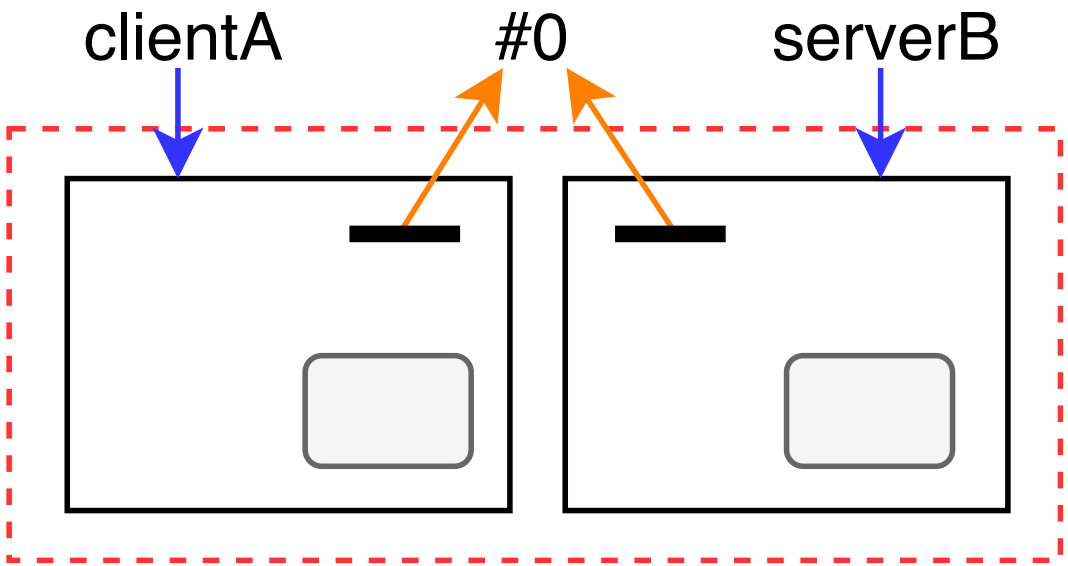
DBCChecker

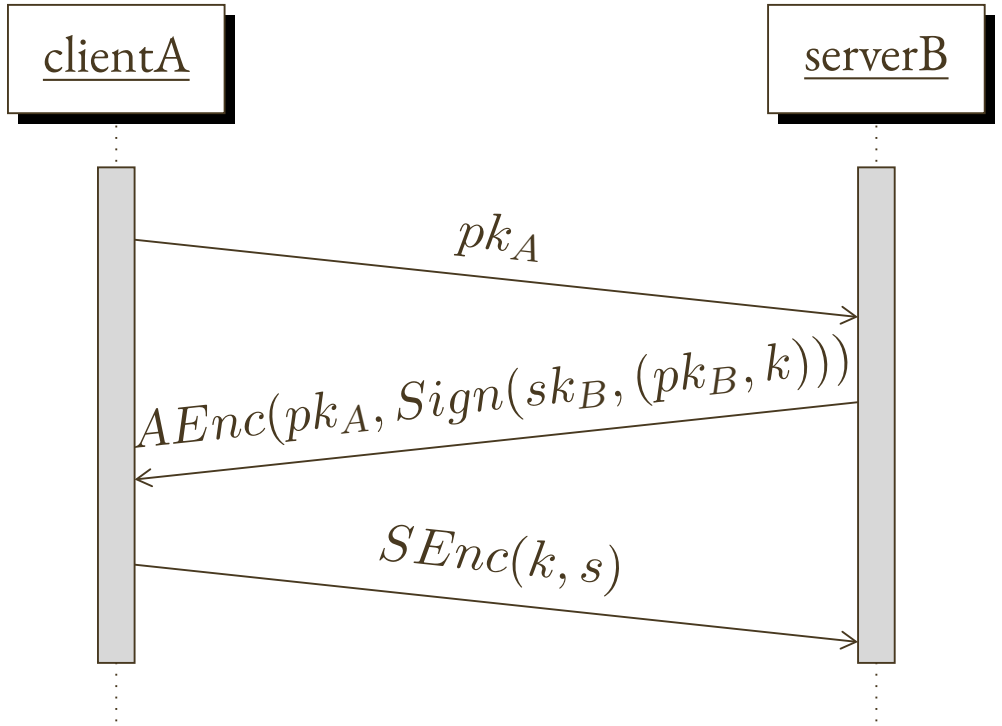
- Input:
  - a configuration of a container-based system
  - for each container, an abstract description of the interaction on its interface (“contract”)
  - Global properties to be checked
- Output: a model for the global system, verifiable in some backend



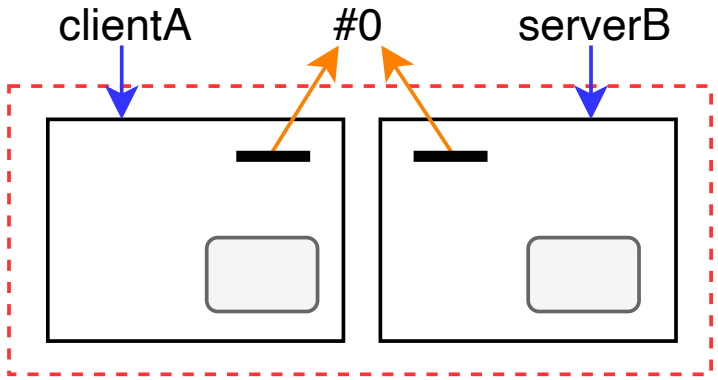
A basic example: secure handshake

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





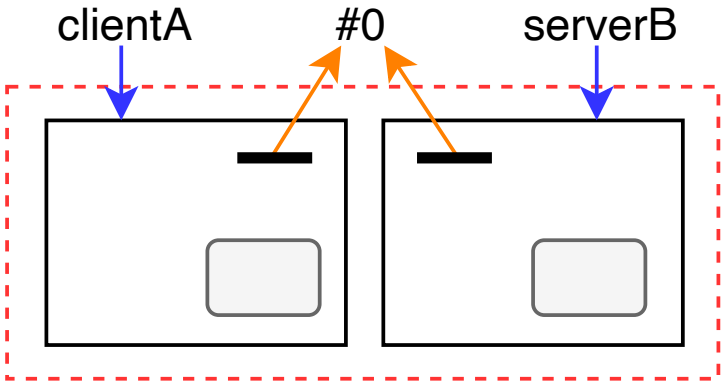
A basic example: secure handshake: contracts

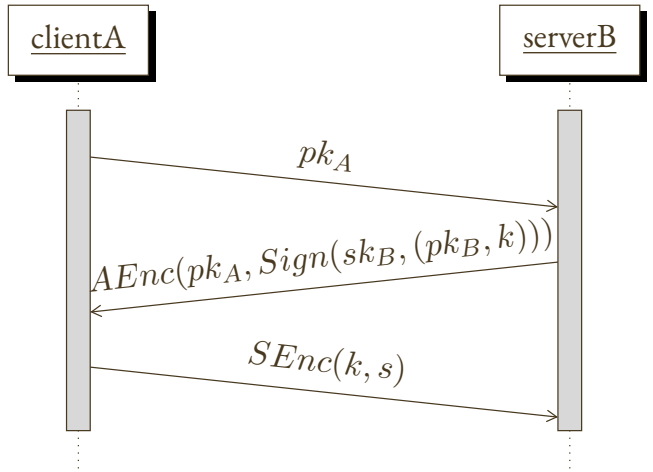


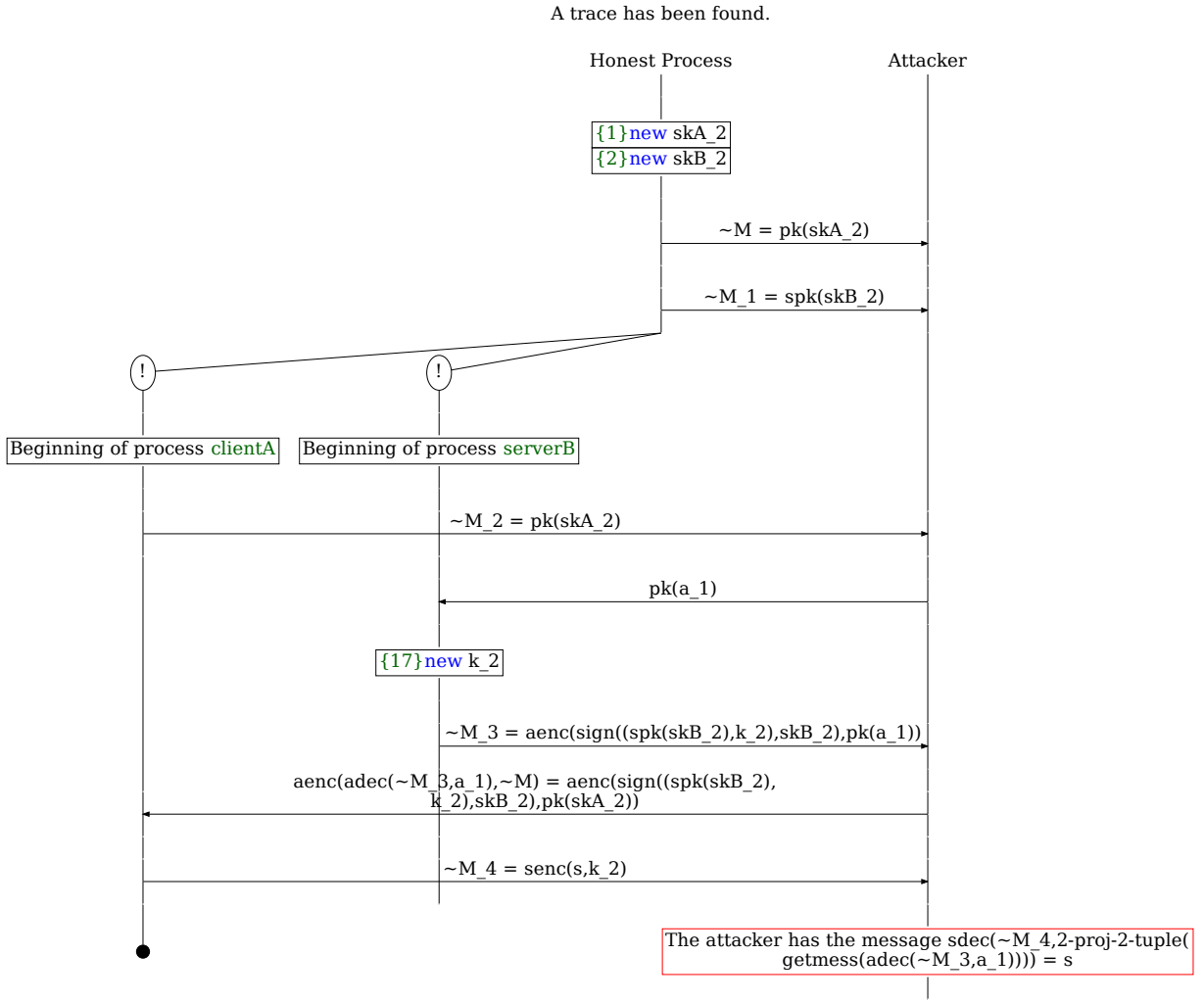

```
"clientA": {
  "metadata": {
    "type": "node",
    "control": "1on0",
    "params": ["pkA:pkey", "skA:skey",
          "pkB:spkey"],
    "behaviour": "!(out (#0+, pkA);
          in (#0+, x : bitstring);
          let y = adec(x, skA) in
          let (=pkB, k : key) = checksign(y,
          pkB)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







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 syntesis
- The sky's is the limit!

Epilogue

- Bigraphs are a well-suited formal metamodel for container-based systems
  - Capture logical connections of components and processes, nesting of components, composition of containers
  - Strong basis for tools and for theoretical results
  - Simple graphical language (amenable also for non experts)
  - Huge possibilities of research projects (e.g. PNRR) and industrial applications

Thanks for your attention! Questions?

marino.miculan@uniud.it