IEEE ICNP 2018

The 26th IEEE International Conference on Network Protocols
Cambridge, UK, September 24-27, 2018

1st P4 European Workshop (P4EU)

Named Data Networking with Programmable Switches

Rui Miguel
(speaker)

LASIGE, Faculdade de Ciências
Universidade de Lisboa
(Faculty of Sciences, University of Lisbon), Lisbon

Salvatore Signorello

SnT,
University of Luxembourg,
Luxembourg

Fernando M. V. Ramos

LASIGE, Faculdade de Ciências
Universidade de Lisboa
(Faculty of Sciences, University of Lisbon), Lisbon

PRESENTATION OUTLINE

I **Background & Motivation**

II **Architecture**
FIB | Pending Interest Table | Content Store

III **Evaluation**

IV **Conclusion & Future Work**

PART I

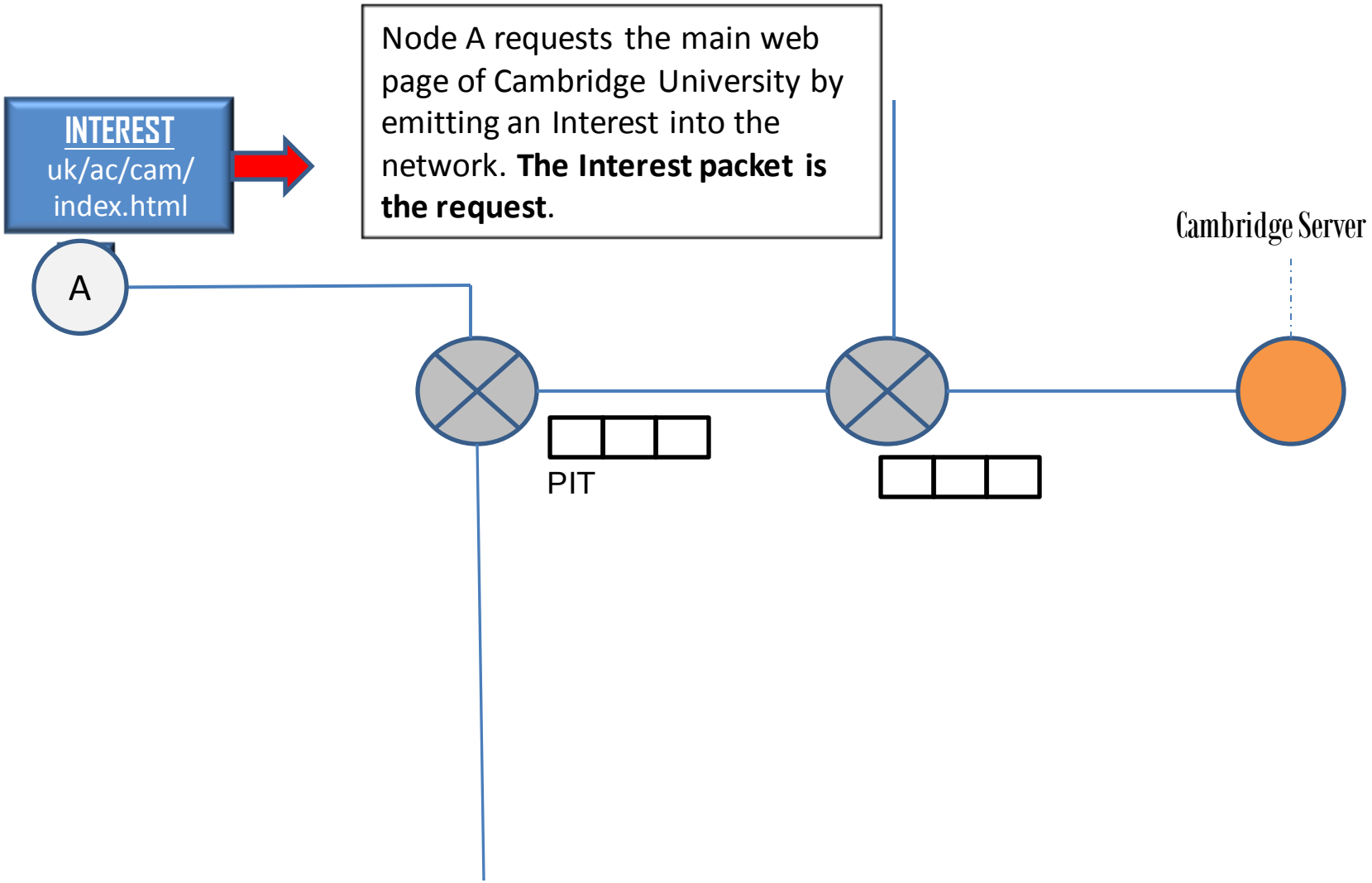
Background

Named Data Networks

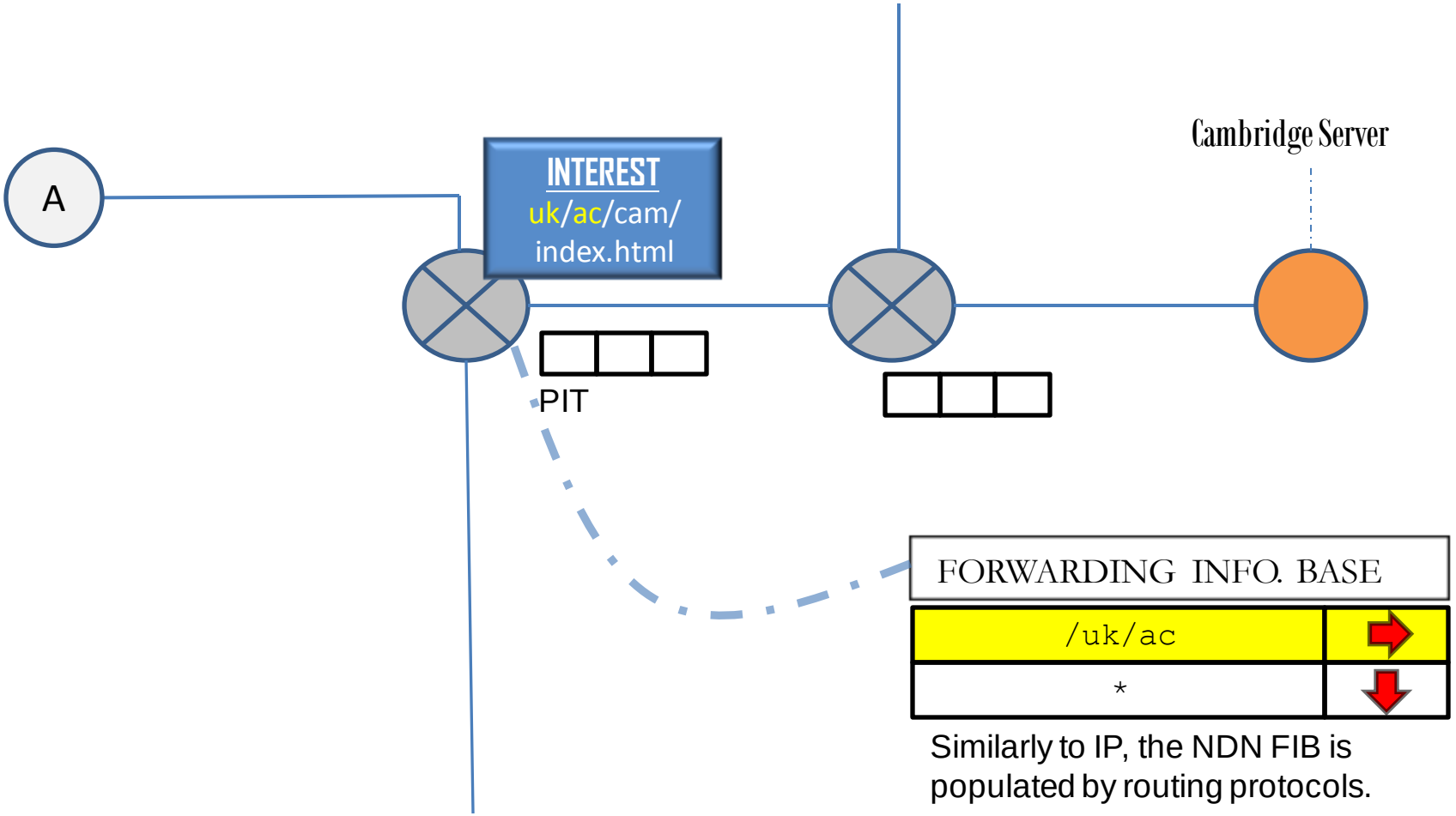
- A network architecture that **focuses on content distribution**.
  - In contrast to the point-to-point IP model.
  - Better suits nowadays Internet's most frequent use cases.

- Machines have no identification (addresses). **Only data is named**.
  - Consumers **request a resource by name** with an Interest packet.
  - **Producers emit a** Data packet uniquely bound to that name.
  - Routers forward these Interests according to its name.
  - Instead of asking a specific host for content, the consumer asks the network.

Named Data Networks

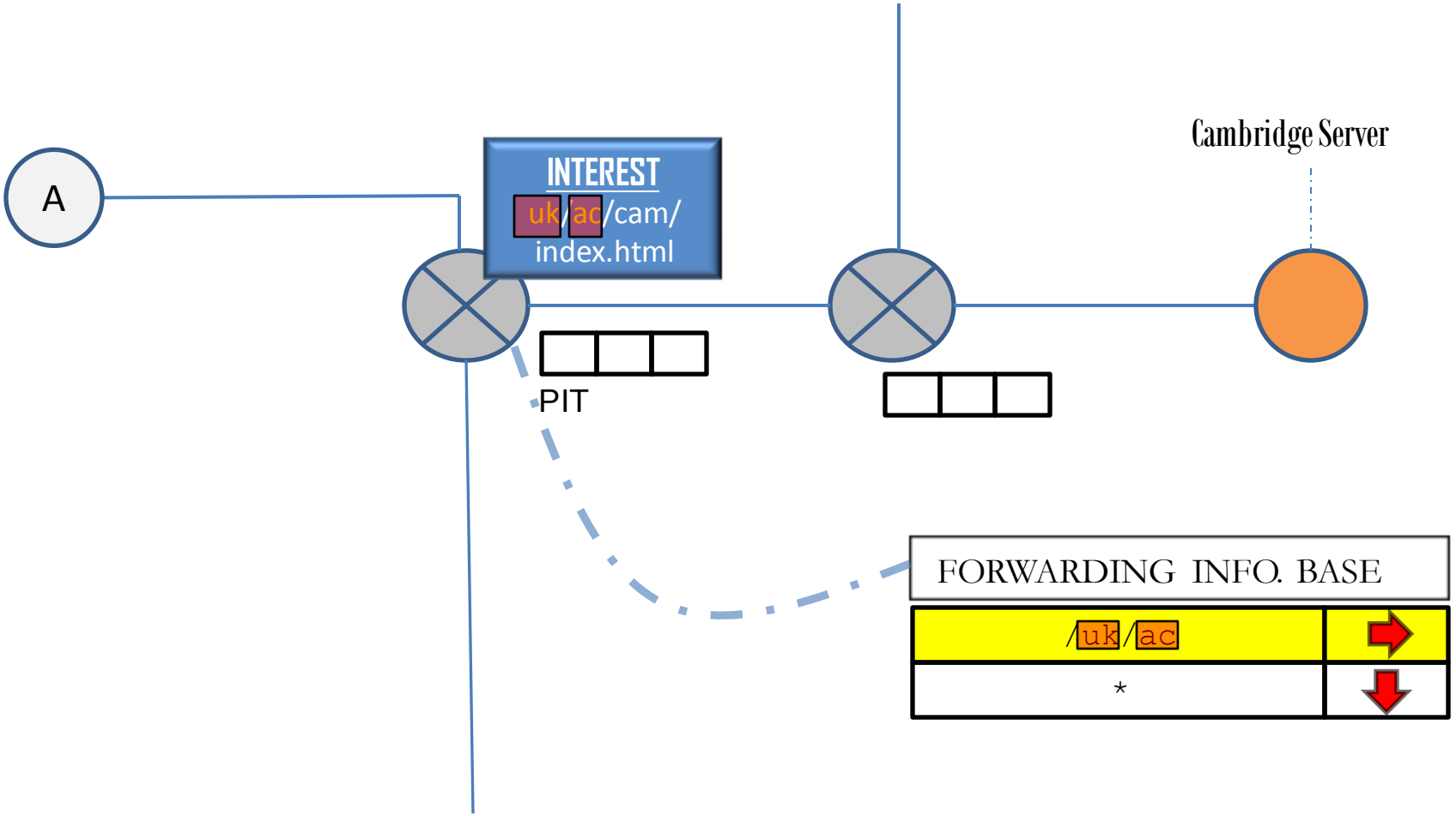


Named Data Networks

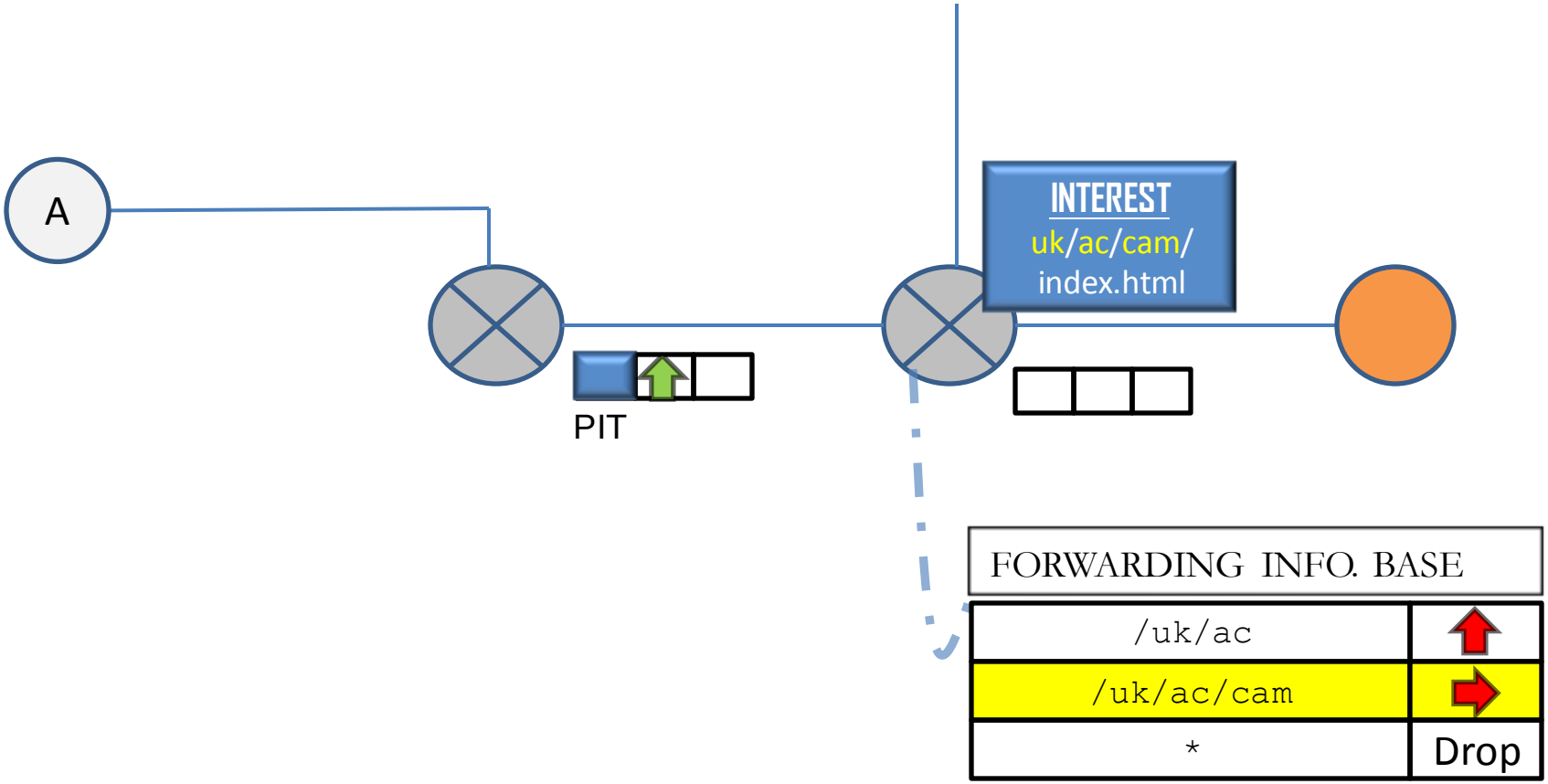


Similarly to IP, the NDN FIB is populated by routing protocols.

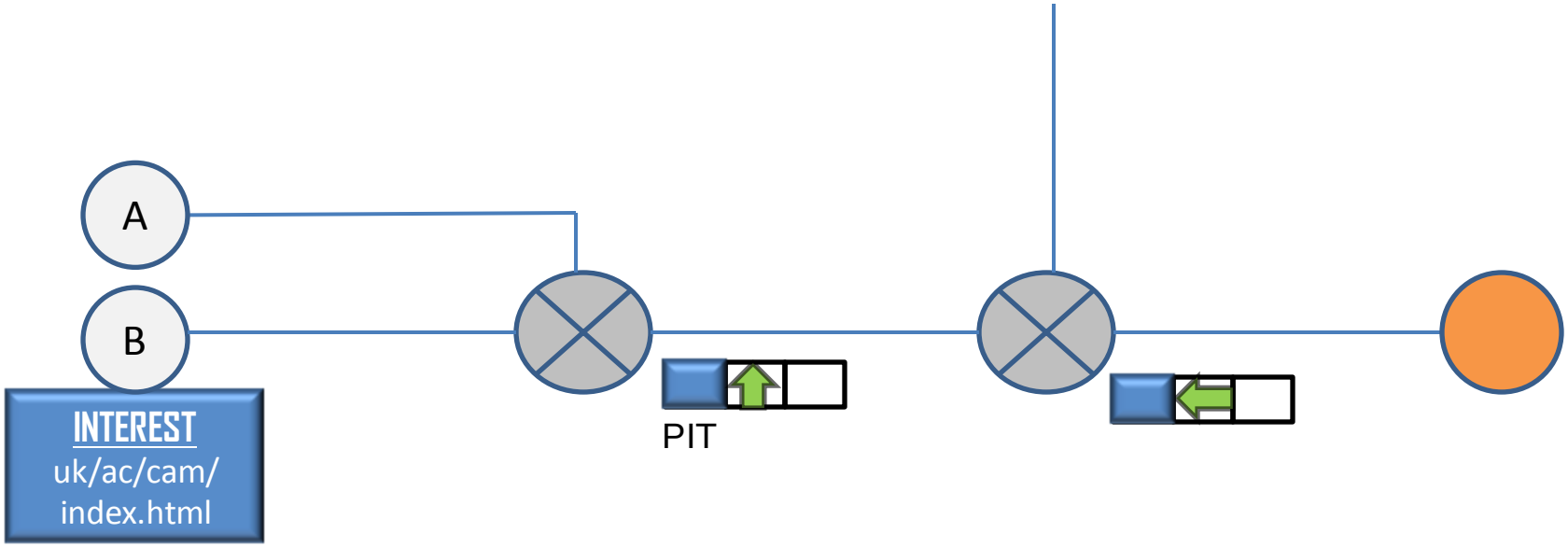
Named Data Networks



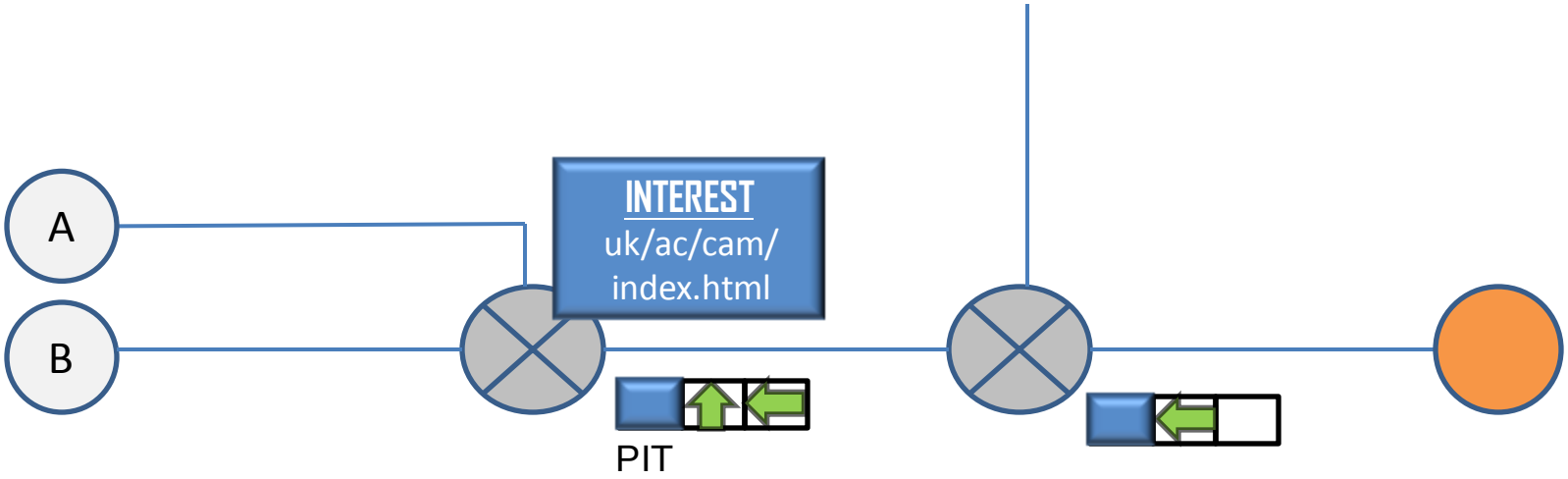
Named Data Networks



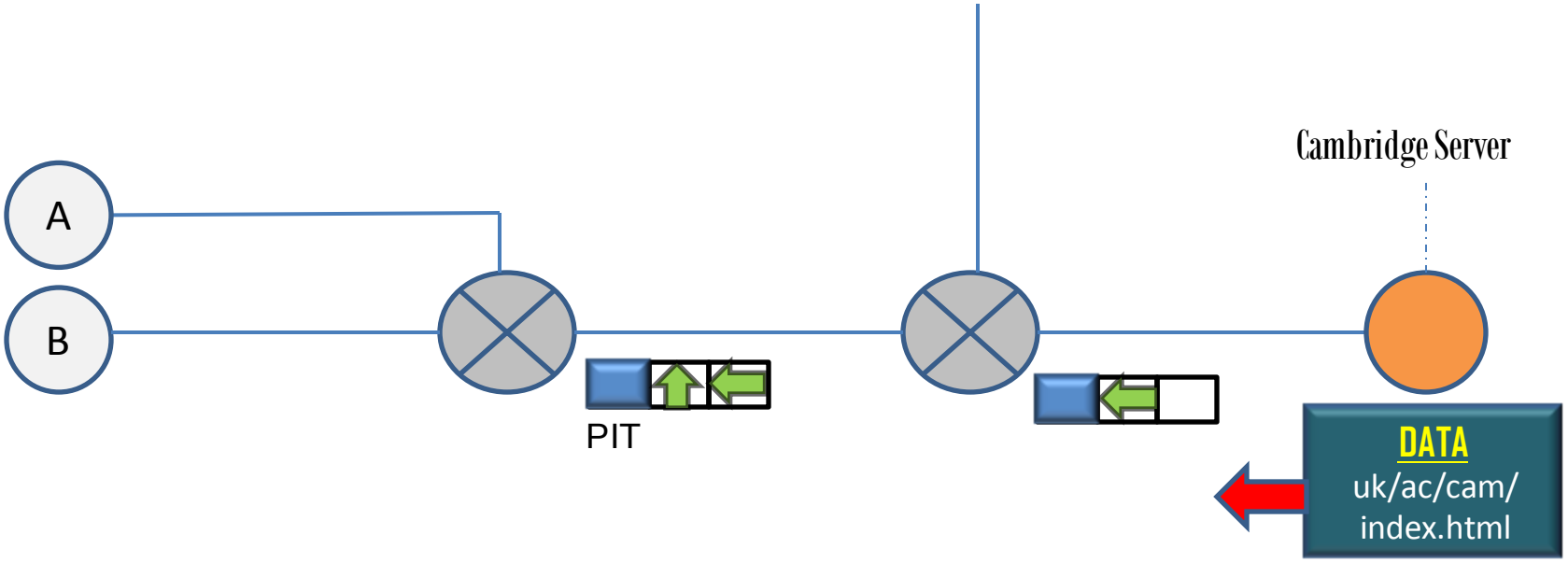
Named Data Networks



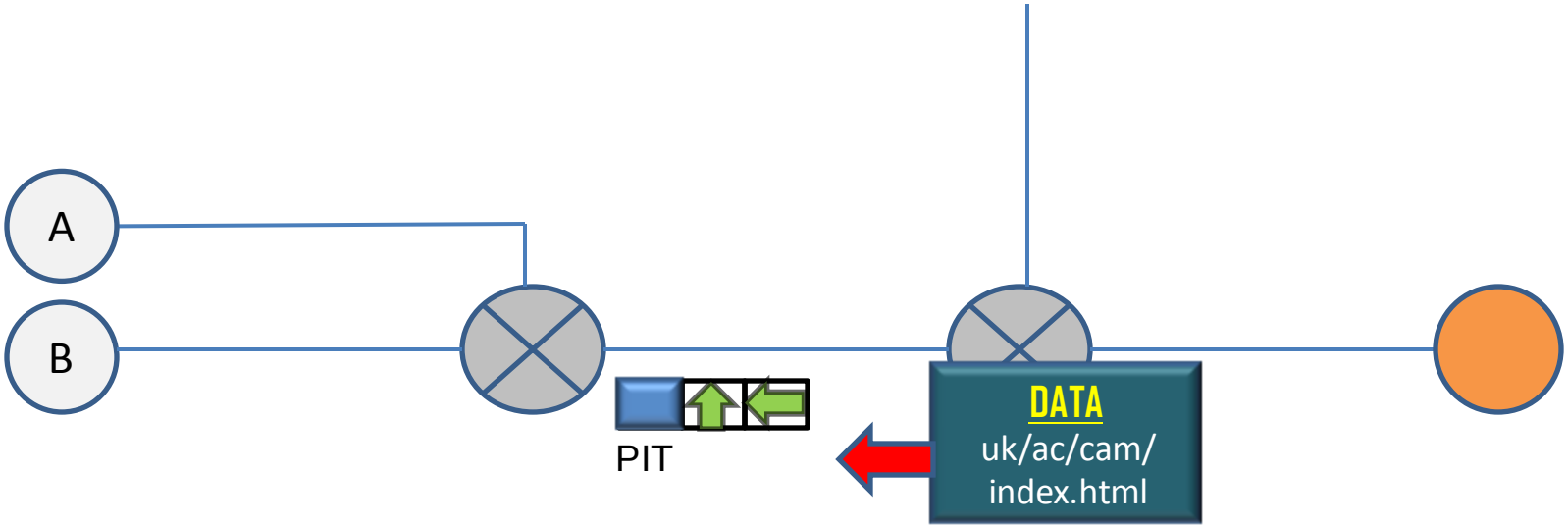
Named Data Networks



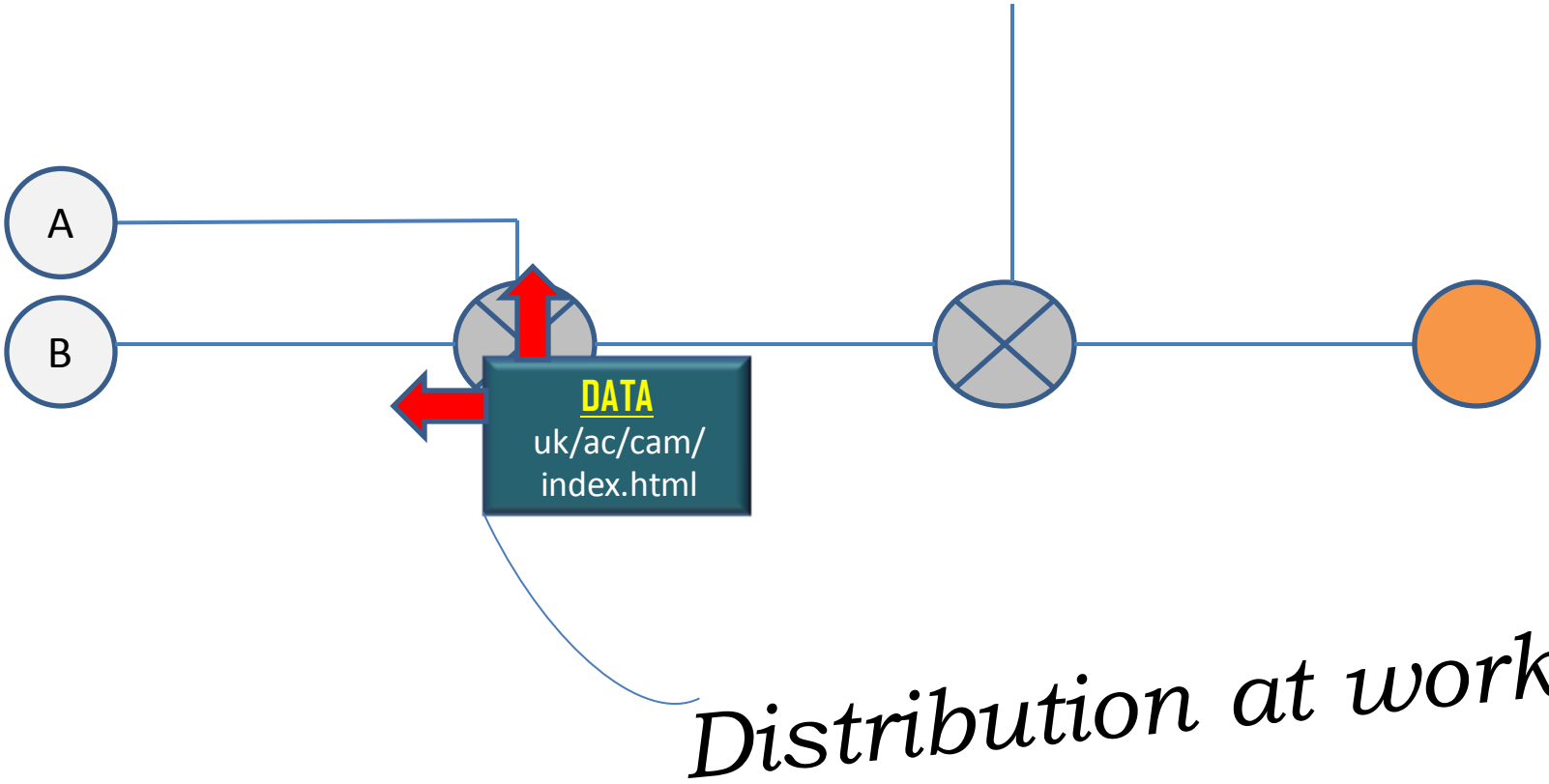
Named Data Networks



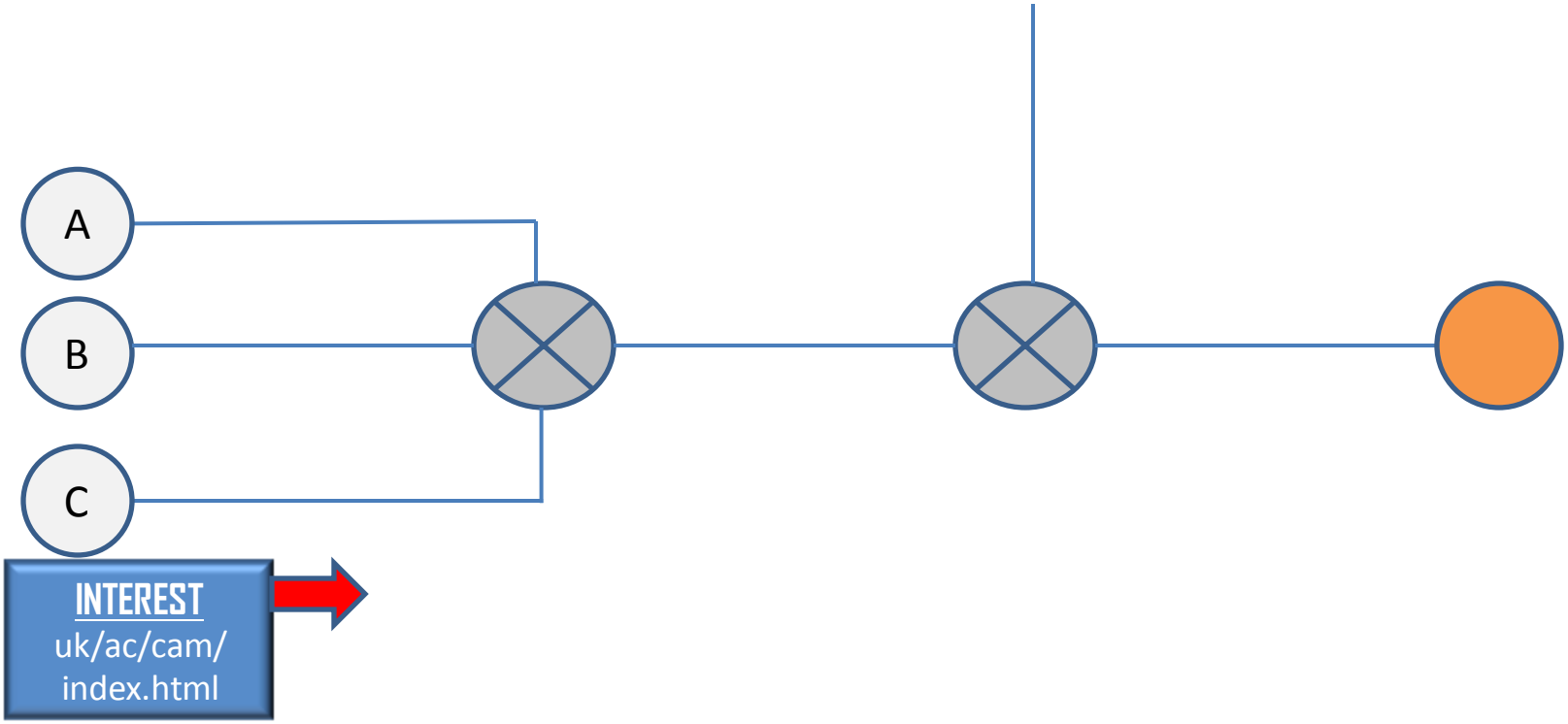
Named Data Networks



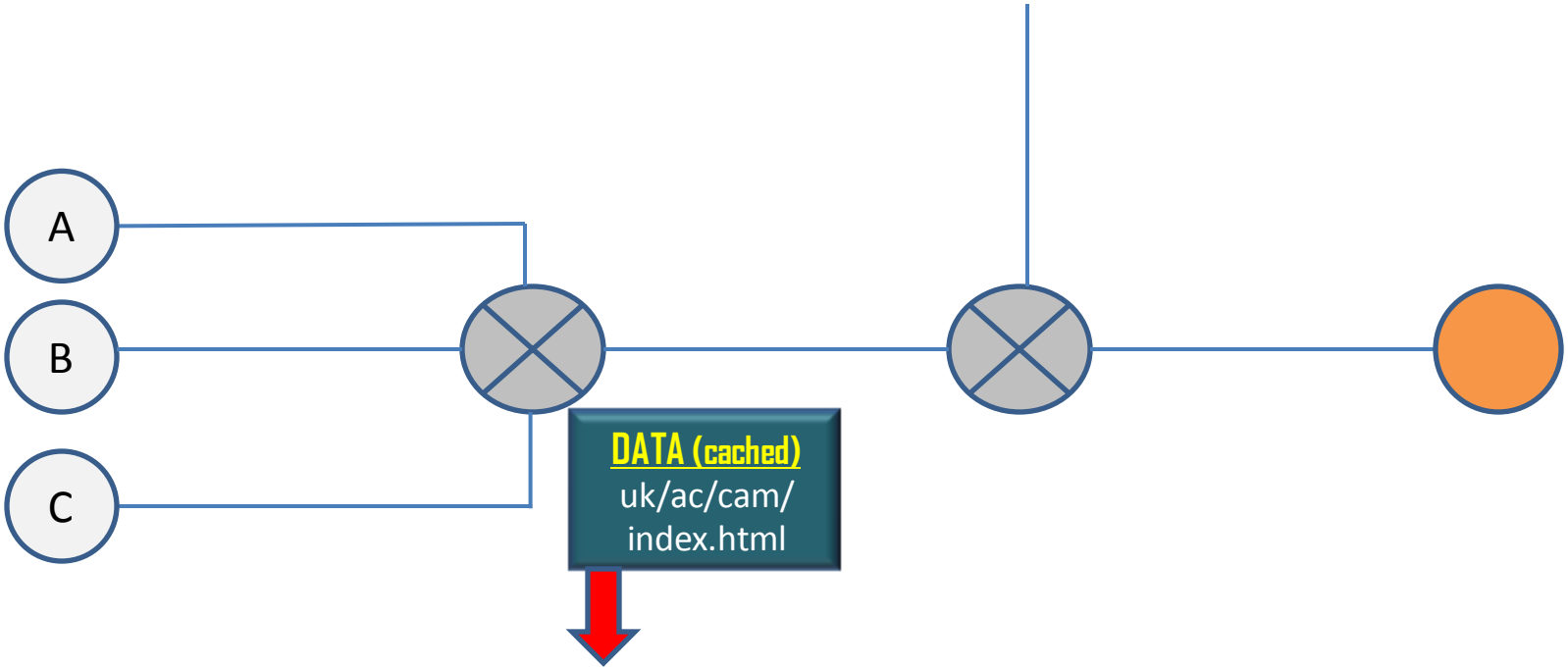
Named Data Networks



Named Data Networks



Named Data Networks



Motivation

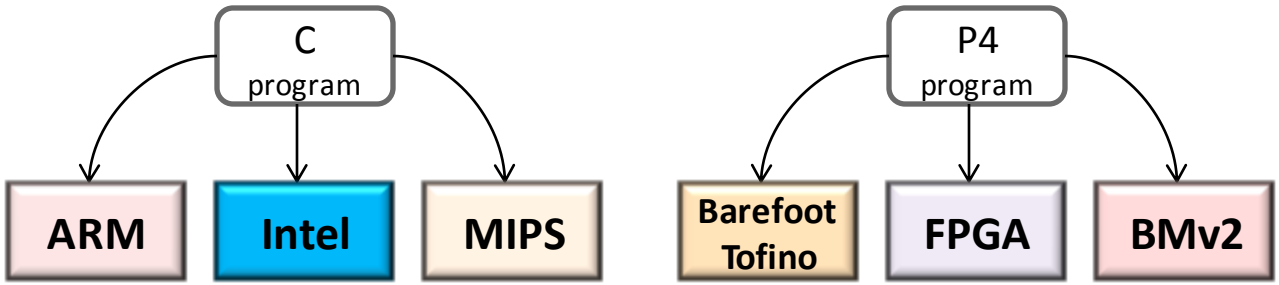
- No line-rate production hardware exists for NDN.
- **Existing routers can't be extended** to support the non-conventional packet processing required by NDN.
  - **Current solutions are software-based**. FIB has thus far always been implemented in software.
- We leverage the **recent availability of programmable switches** to propose the design and implementation of a new line-rate NDN router written in P4_16.

PART II

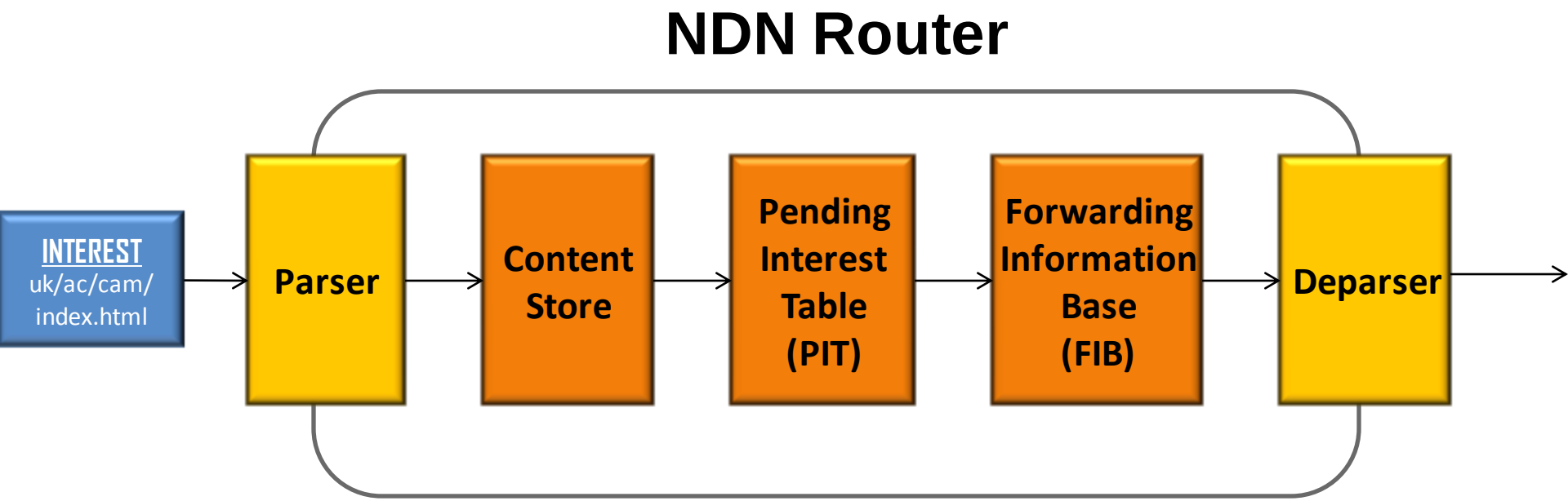
Architecture

$P4_{16}$ as implementation language

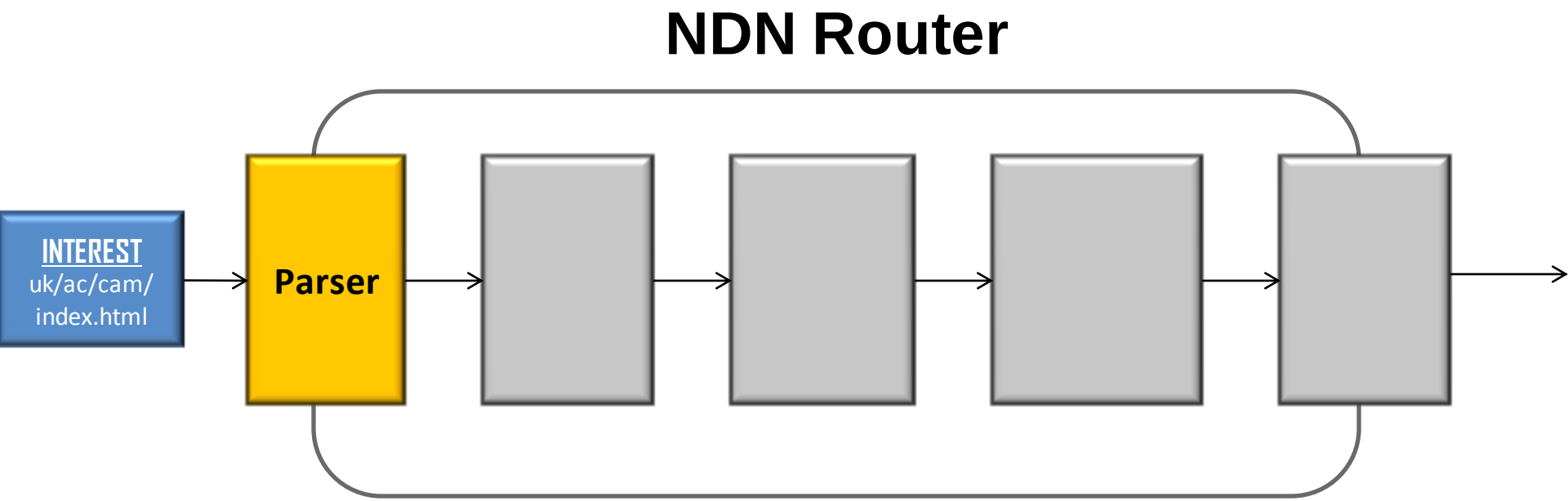
- High-level **language to express data plane** forwarding behavior.
- The programmer specifies **headers, parser and the processing sequence** a packet should undergo.
- **A compiler maps the program** to the underlying device's capabilities and hardware.



Architecture

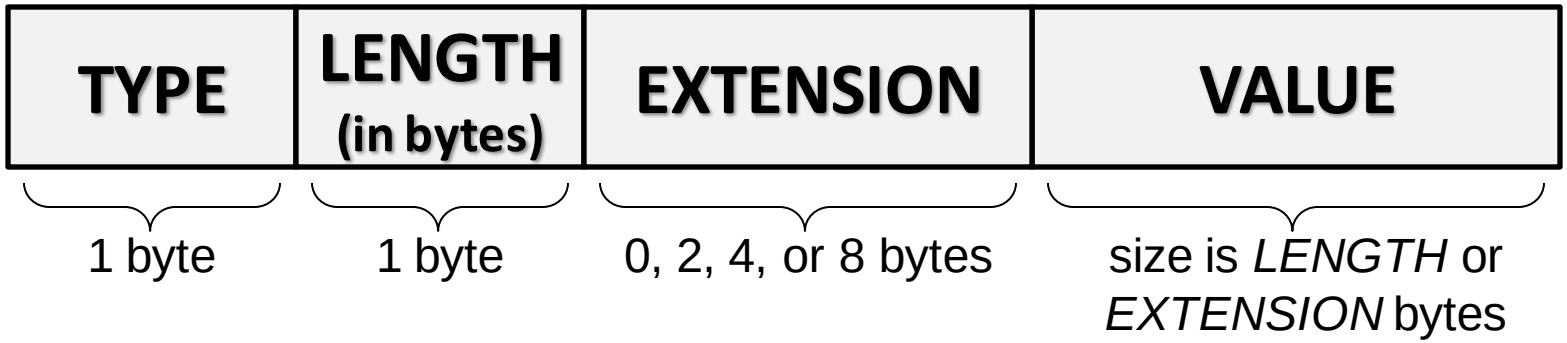


Architecture



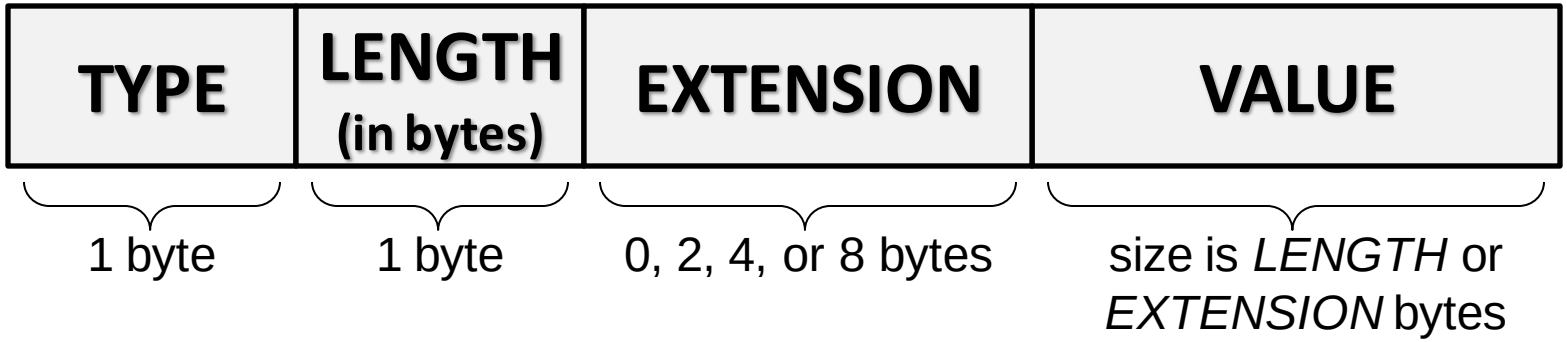
Parser

- **PROBLEM 1:** NDN packets do not follow the typical packet structure. They are a tree of **Type-Length-Values (TLVs)**.

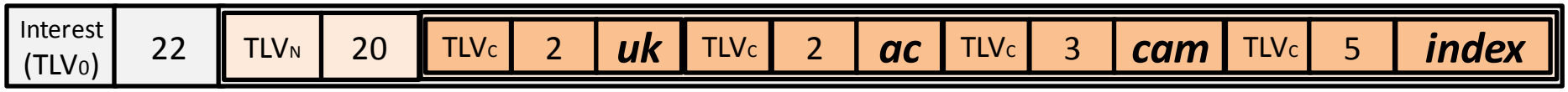


Parser

- **PROBLEM 1:** NDN packets do not follow the typical packet structure. They are a tree of **Type-Length-Values (TLVs)**.



- What “uk/ac/cam/index” looks like at the network level:



Parser

- In P4_14, parsing this packet structure incurs an enormous amount of parser states.

Parser

- In P4_14, parsing this packet structure incurs an enormous amount of parser states.

- P4_16 makes it easier. **SOLUTION:**
  1. Use a **header_union**. The union's members cover all the TLV possibilities.
  2. Encapsulate TLV extraction logic inside a **subparser**. The main **parser** calls the subparser whenever a TLV needs to be extracted from the packet.

```
header smallTLV_h {
 bit<8> type;
 bit<8> length;
 varbit<252 × 8> value;
}
```

```
header mediumTLV_h {
 bit<8> type;
 bit<8> lencode; //=253
 bit<16> extension;
 varbit<0xffff × 8> value;
}
```

```
header_union tlv_hu {
 smallTLV_h stlv;
 mediumTLV_h mtlv;
 largeTLV_h ltlv;
}
```

Parser

- In P4_14, parsing this packet structure incurs an enormous amount of parser states.

- P4_16 makes it easier. **SOLUTION:**
  1. Use a **header_union**. The union's members cover all the TLV possibilities.
  2. Encapsulate TLV extraction logic inside a **subparser**. The main **parser** calls the subparser whenever a TLV needs to be extracted from the packet.

- **ADVANTAGES:**
  + P4 program with less code & more readable

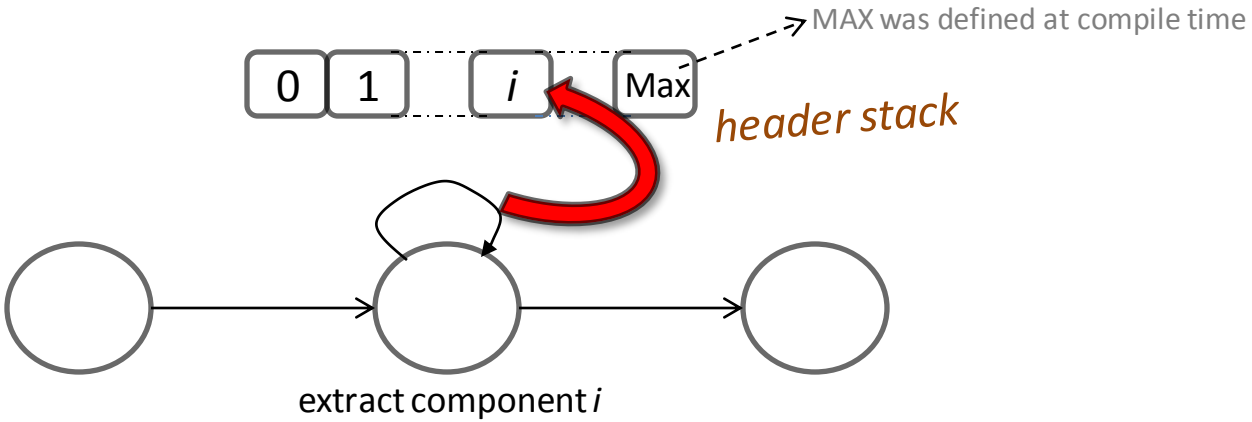
```
header smallTLV_h {
 bit<8> type;
 bit<8> length;
 varbit<252 × 8> value;
}
```

```
header mediumTLV_h {
 bit<8> type;
 bit<8> lencode; //=253
 bit<16> extension;
 varbit<0xffff × 8> value;
}
```

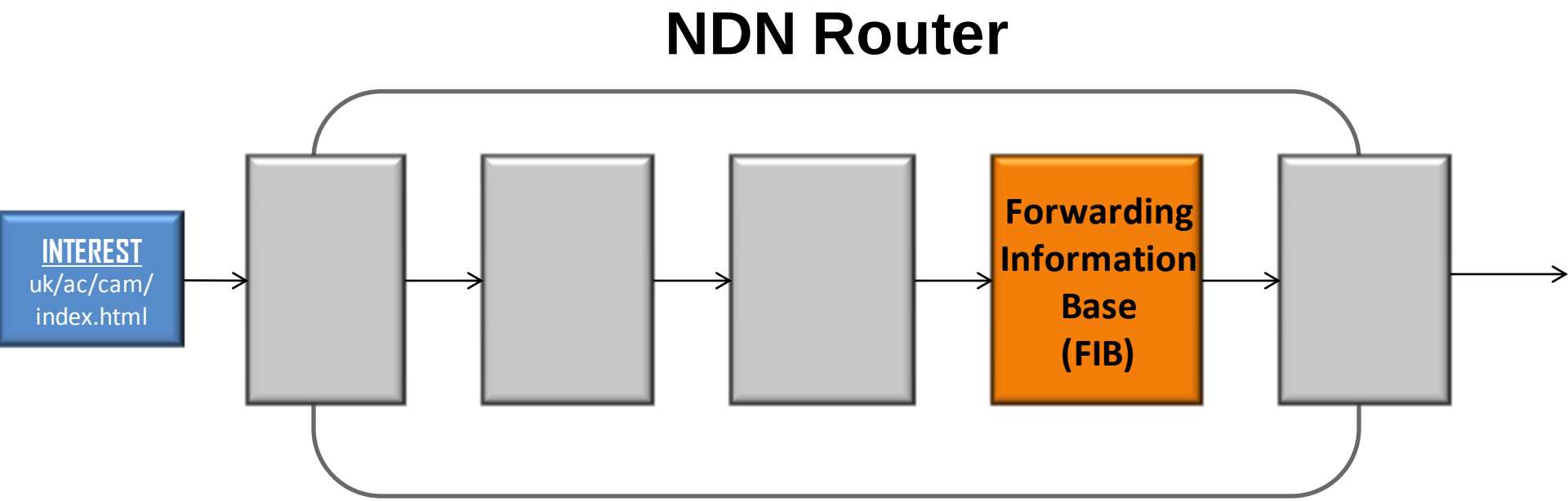
```
header_union tlv_hu {
 smallTLV_h stlv;
 mediumTLV_h mtlv;
 largeTLV_h ltlv;
}
```

Parser

- **PROBLEM 2:** Interest names may have any number of components.
  - "uk/ac/cam/index" (4 components)
- **SOLUTION:**
  - Use the **header stack** P4 type.
  - The parser is a state machine that can transition to itself.
  - BMv2-ss limitation: one must write MAX parser states, because only compile-time values are allowed as header stack indexes.



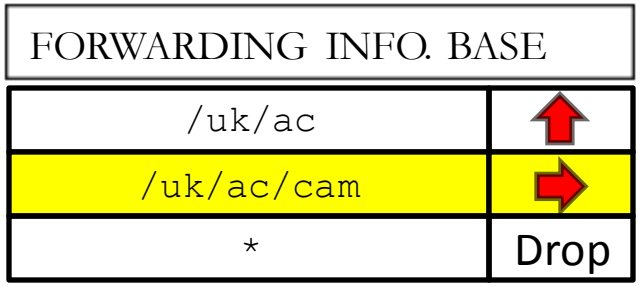
Architecture



Forwarding Information Base (FIB)

- The **FIB** routes an Interest packet by longest prefix match of the name therein.

- **PROBLEM:** P4 has little support to process strings. We can parse them to **varbit** fields, but these can't be used to match on tables.

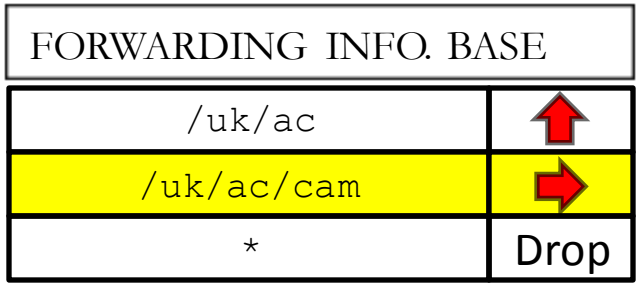


Forwarding Information Base (FIB)

- The **FIB** routes an Interest packet by longest prefix match of the name therein.

- **PROBLEM:** P4 has little support to process strings. We can parse them to **varbit** fields, but these can't be used to match on tables.

- **SOLUTION:**
  1. Use the hash() primitive to convert to an unsigned, fixed-length **bit** type.
  2. Perform lpm (longest prefix match) **match kind** on the result (details follow).



FIB

- Introducing a new data structure, the **hashtray**.
  - **DEFINITION:** Given an NDN name, a hashtray is a series of blocks each with the result of hashing a component.
  - In our implementation, we used MAX=8 blocks and a 16-bit hash function (names longer than 8 components match only with the first 8 components).

```
typedef bit<8 × 16> hashtray_t;
```

- The hashtray is used in 2 situations:

1. When building FIB entries. Each FIB entry is a hashtray.

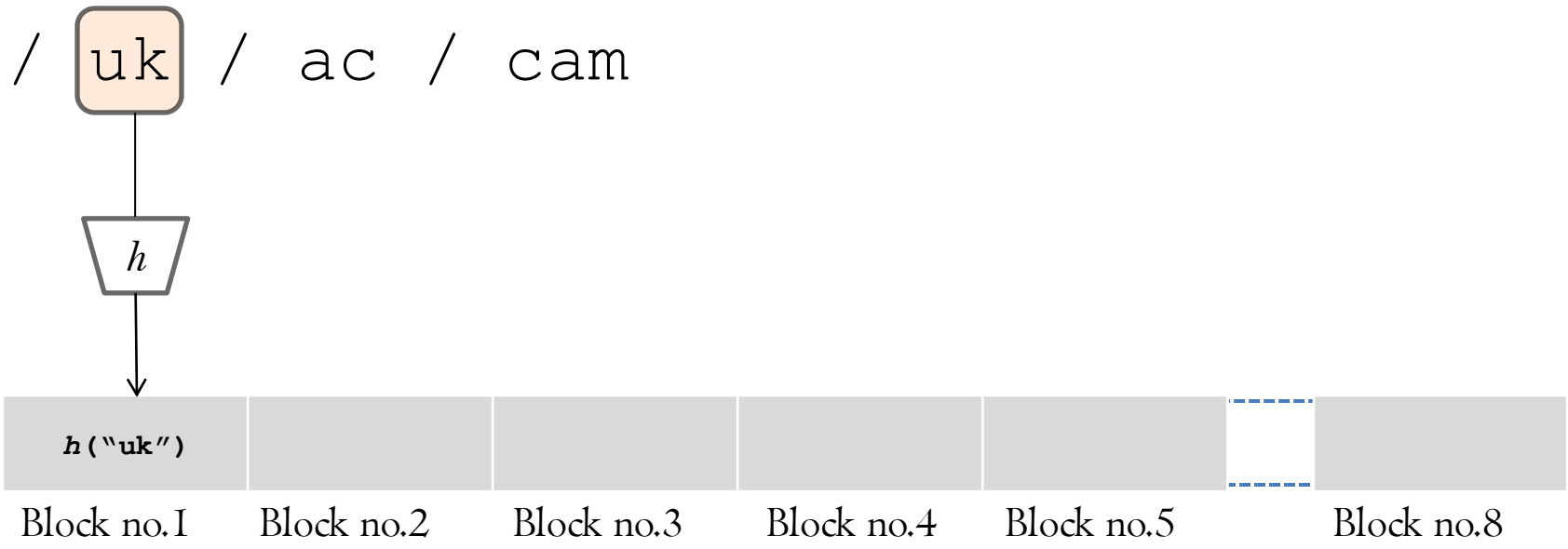
2. Whenever an Interest needs to be routed.

Constructing the FIB

- Let's add the entry: `/ uk / ac / cam → port 2`

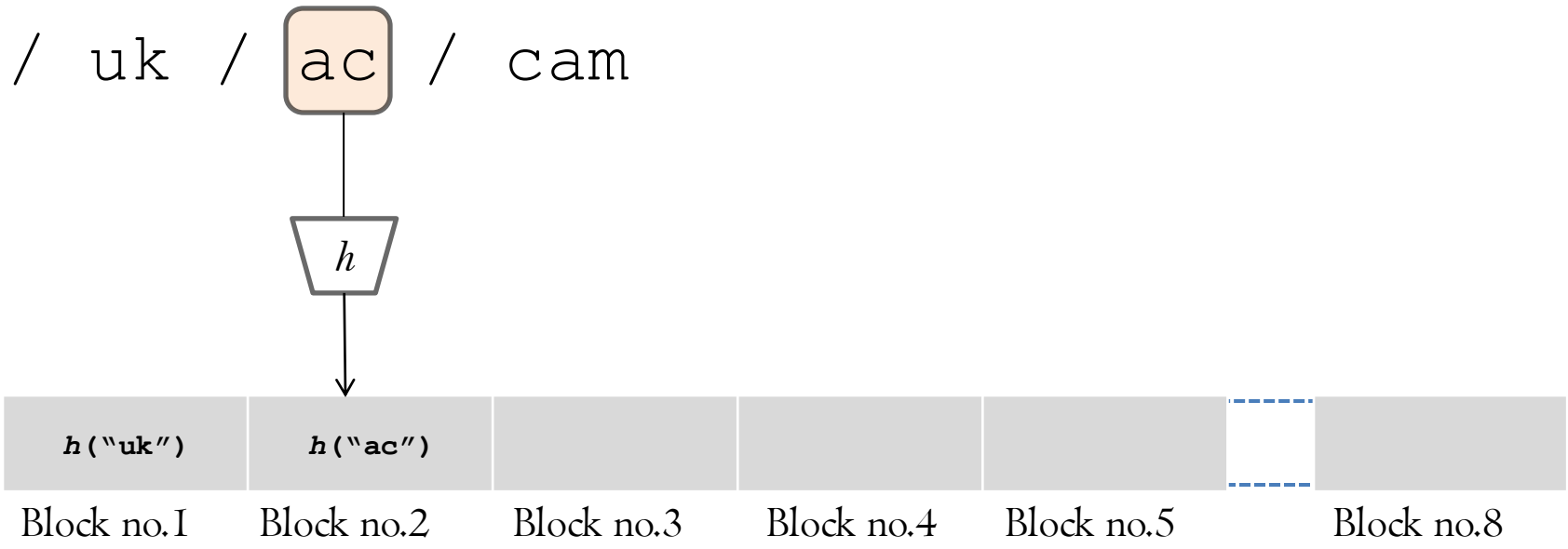
Constructing the FIB

- Let's add the entry:



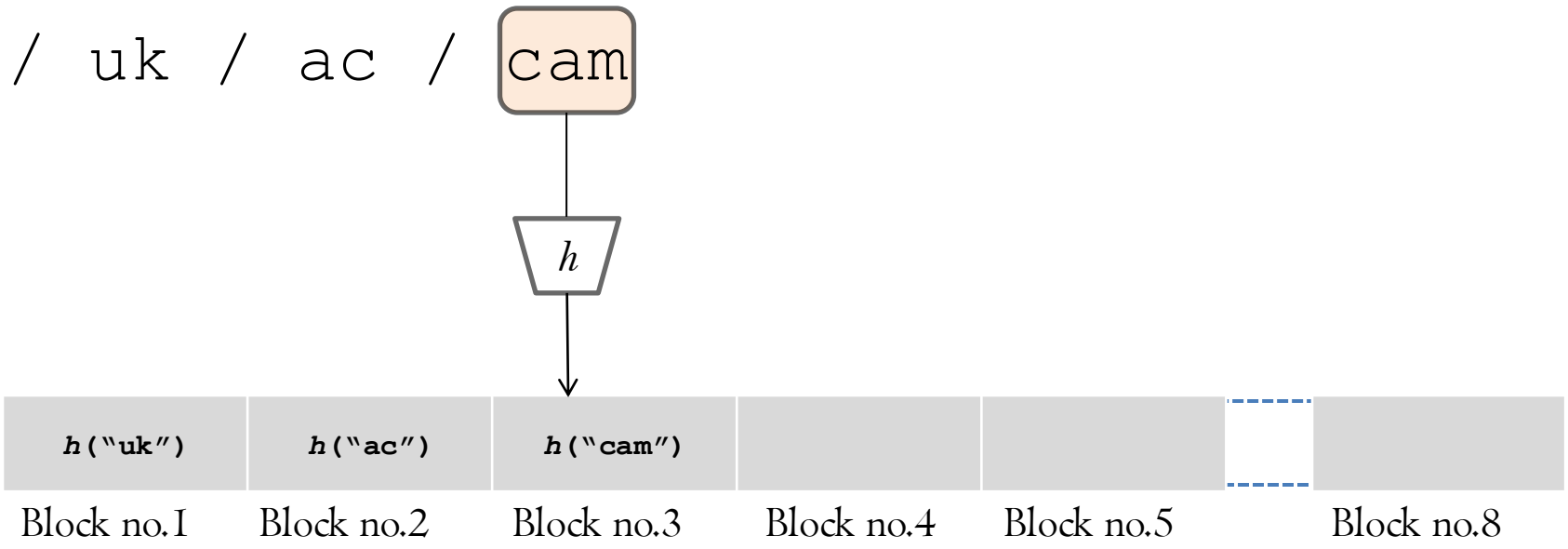
Constructing the FIB

- Let's add the entry:



Constructing the FIB

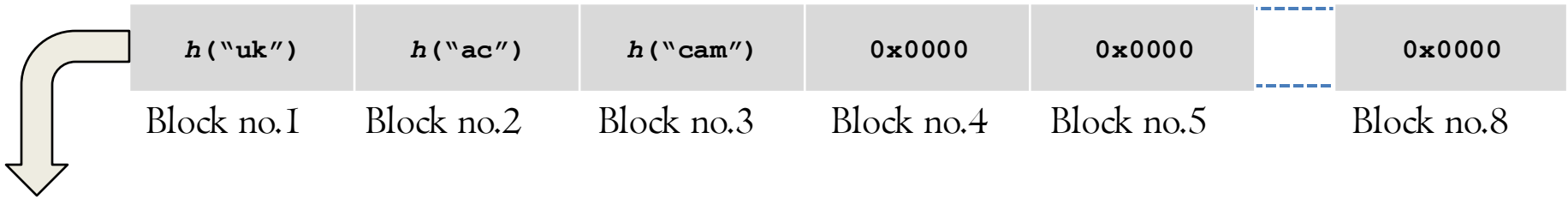
- Let's add the entry:



Constructing the FIB

- Let's add the entry:

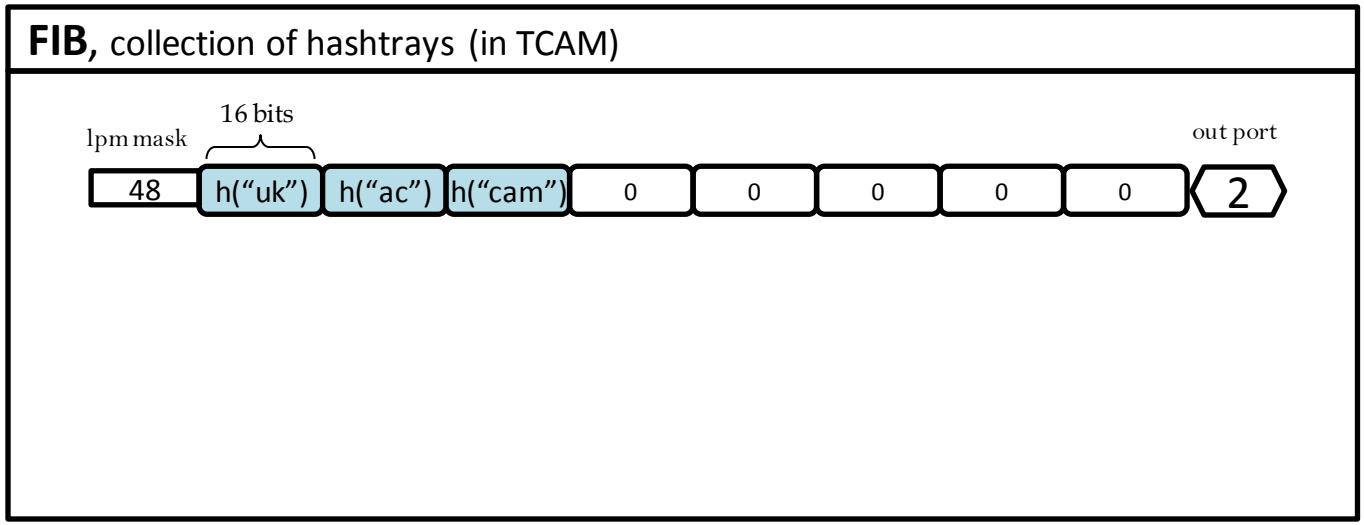
```
/ uk / ac / cam
```



Add hashtray to the FIB

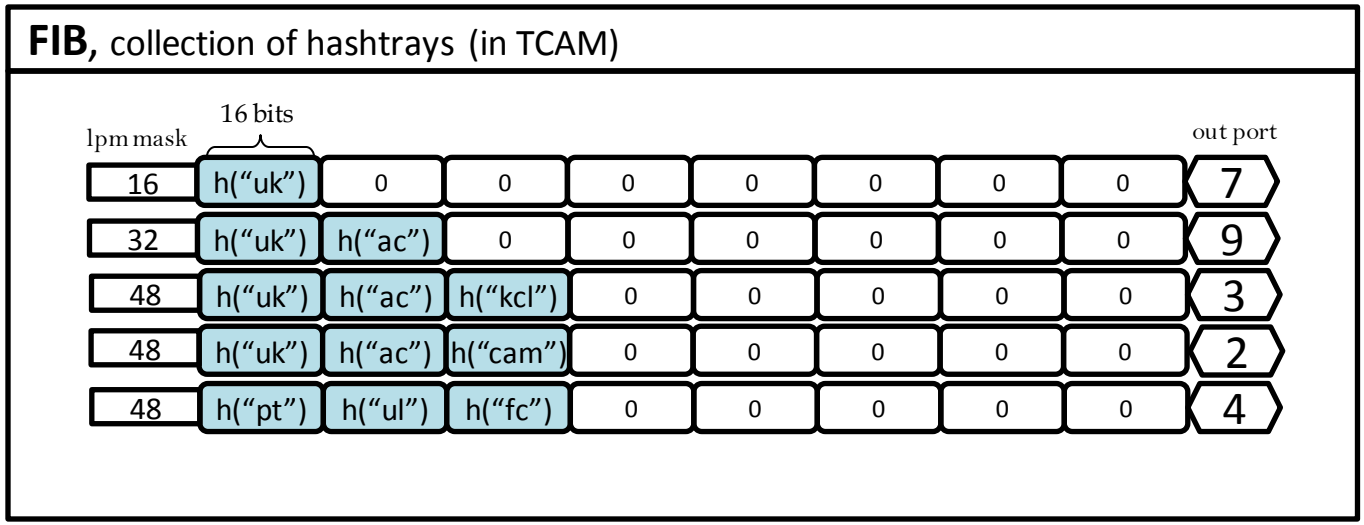
Constructing the FIB

- Our entry is associated with an lpm mask covering three blocks (48 bits), as well as the outgoing port.

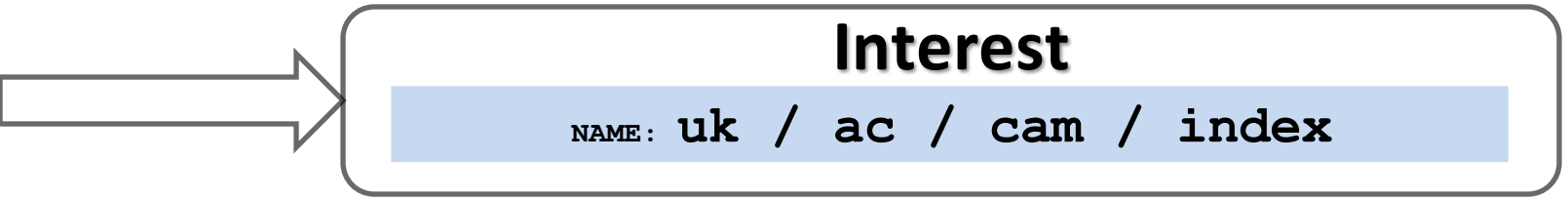


Constructing the FIB

- All entries are added using the same process.



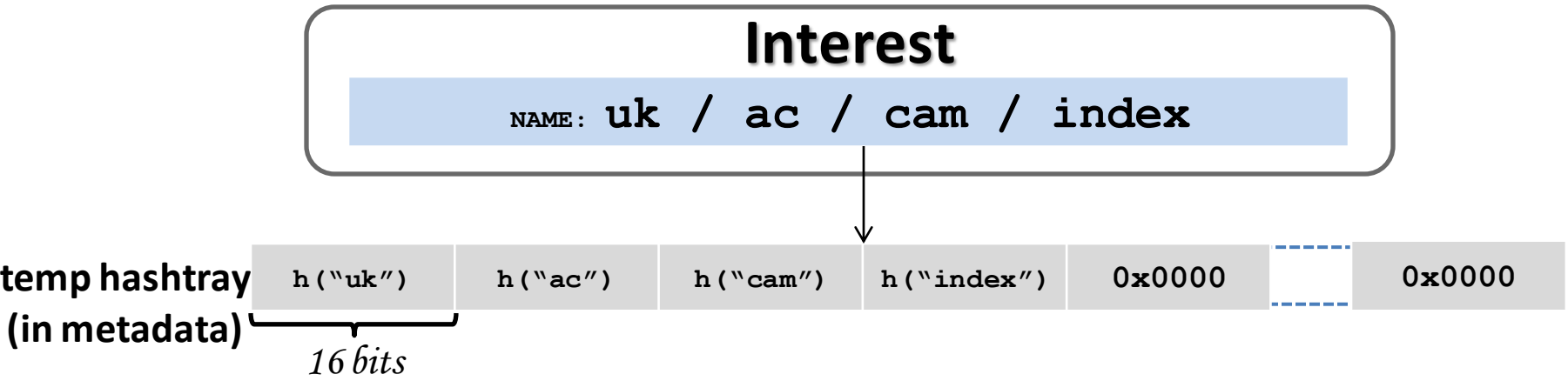
FIB lpm matching



FIB, collection of hashtrays (in TCAM)

lpm mask	16 bits								out port
16	h("uk")	0	0	0	0	0	0	0	7
32	h("uk")	h("ac")	0	0	0	0	0	0	9
48	h("uk")	h("ac")	h("kcl")	0	0	0	0	0	3
48	h("uk")	h("ac")	h("cam")	0	0	0	0	0	2
48	h("pt")	h("ul")	h("fc")	0	0	0	0	0	4

FIB lpm matching



FIB, collection of hashtrays (in TCAM)

lpm mask	16 bits								out port
16	h("uk")	0	0	0	0	0	0	0	7
32	h("uk")	h("ac")	0	0	0	0	0	0	9
48	h("uk")	h("ac")	h("kcl")	0	0	0	0	0	3
48	h("uk")	h("ac")	h("cam")	0	0	0	0	0	2
48	h("pt")	h("ul")	h("fc")	0	0	0	0	0	4

FIB lpm matching

Interest

NAME: **uk / ac / cam / index**

temp hashtray (in metadata)

h("uk")	h("ac")	h("cam")	h("index")	0x0000		0x0000

16 bits

fib.apply()

FIB, collection of hashtrays (in TCAM)

lpm mask	16 bits								out port
16	h("uk")	0	0	0	0	0	0	0	7
32	h("uk")	h("ac")	0	0	0	0	0	0	9
48	h("uk")	h("ac")	h("kcl")	0	0	0	0	0	3
48	h("uk")	h("ac")	h("cam")	0	0	0	0	0	2
48	h("pt")	h("ul")	h("fc")	0	0	0	0	0	4

FIB lpm matching

Interest

NAME: **uk / ac / cam / index**

temp hashtray (in metadata)

h("uk")	h("ac")	h("cam")	h("index")	0x0000		0x0000

16 bits

fib.apply()

FIB, collection of hashtrays (in TCAM)

lpm mask	16 bits								out port
16	h("uk")	0	0	0	0	0	0	0	7
32	h("uk")	h("ac")	0	0	0	0	0	0	9
48	h("uk")	h("ac")	h("kcl")	0	0	0	0	0	3
48	h("uk")	h("ac")	h("cam")	0	0	0	0	0	2
48	h("pt")	h("ul")	h("fc")	0	0	0	0	0	4

FIB lpm matching

Interest

NAME: **uk / ac / cam / index**

temp hashtray (in metadata)

h("uk")	h("ac")	h("cam")	h("index")	0x0000		0x0000

16 bits

fib.apply()

FIB, collection of hashtrays (in TCAM)

lpm mask	16 bits								out port
16	h("uk")	0	0	0	0	0	0	0	7
32	h("uk")	h("ac")	0	0	0	0	0	0	9
48	h("uk")	h("ac")	h("kcl")	0	0	0	0	0	3
48	h("uk")	h("ac")	h("cam")	0	0	0	0	0	2
48	h("pt")	h("ul")	h("fc")	0	0	0	0	0	4

FIB lpm matching

Interest

NAME: **uk / ac / cam / index**

temp hashtray (in metadata)

h("uk")	h("ac")	h("cam")	h("index")	0x0000		0x0000

16 bits

fib.apply()

FIB, collection of hashtrays (in TCAM)

lpm mask	16 bits								out port
16	h("uk")	0	0	0	0	0	0	0	7
32	h("uk")	h("ac")	0	0	0	0	0	0	9
48	h("uk")	h("ac")	h("kcl")	0	0	0	0	0	3
48	h("uk")	h("ac")	h("cam")	0	0	0	0	0	2
48	h("pt")	h("ul")	h("fc")	0	0	0	0	0	4

FIB lpm matching

Interest

NAME: **uk / ac / cam / index**

temp hashtray (in metadata)

h("uk")	h("ac")	h("cam")	h("index")	0x0000		0x0000

16 bits

fib.apply()

FIB, collection of hashtrays (in TCAM)

	lpm mask	16 bits								out port
✓	16	h("uk")	0	0	0	0	0	0	0	7
✓	32	h("uk")	h("ac")	0	0	0	0	0	0	9
✗	48	h("uk")	h("ac")	h("kcl")	0	0	0	0	0	3
→	48	h("uk")	h("ac")	h("cam")	0	0	0	0	0	2
	48	h("pt")	h("ul")	h("fc")	0	0	0	0	0	4

FIB lpm matching

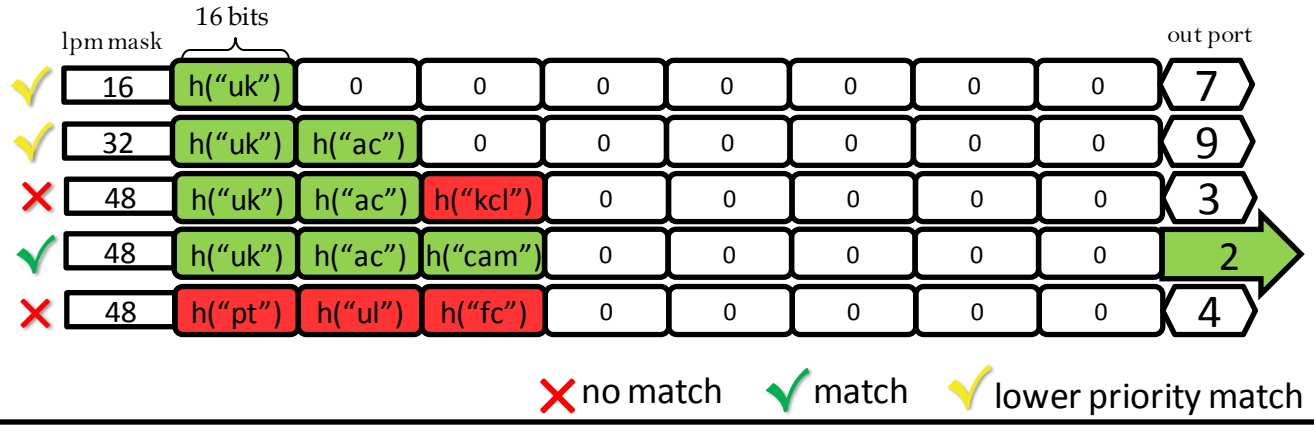
Interest

NAME: **uk / ac / cam / index**

temp hashtray (in metadata)	h("uk")	h("ac")	h("cam")	h("index")	0x0000		0x0000

16 bits

FIB, collection of hashtrays (in TCAM)



FIB lpm matching

Interest

NAME: **uk / ac / cam / index**

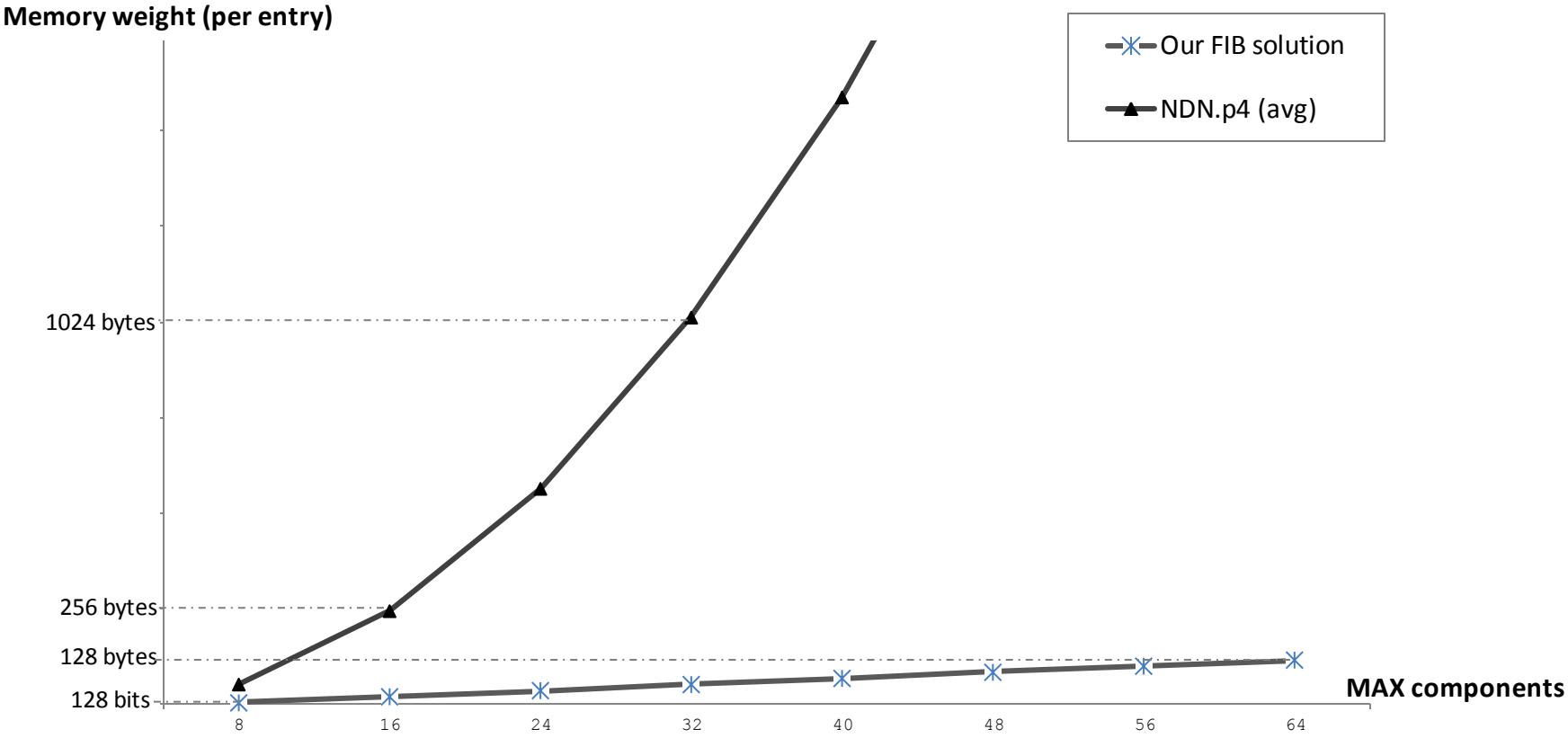
port 2

FIB, collection of hashtrays (in TCAM)

lpm mask	16 bits								out port
16	h("uk")	0	0	0	0	0	0	0	7
32	h("uk")	h("ac")	0	0	0	0	0	0	9
48	h("uk")	h("ac")	h("kcl")	0	0	0	0	0	3
48	h("uk")	h("ac")	h("cam")	0	0	0	0	0	2
48	h("pt")	h("ul")	h("fc")	0	0	0	0	0	4

FIB lpm matching

- Our method guarantees line-rate processing and **greatly reduces memory footprint** over the state-of-the-art.

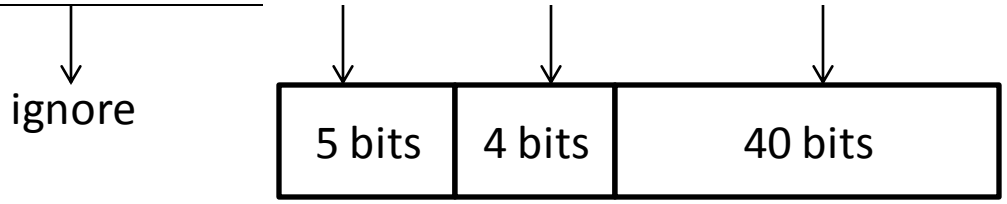


FIB lpm matching

- Our method **is highly flexible**.
  - Not all components need to be used to build the hashtray;
  - Blocks need not necessarily be the same size;
  - So long as the FIB entries and temporary hashtrays from passing Interests are built the same way, the scheme works.

EXAMPLE:

- Doesn't start with "uk/ac/cam" → port 1
- /uk/ac/cam/dpt/group/resource → consult FIB

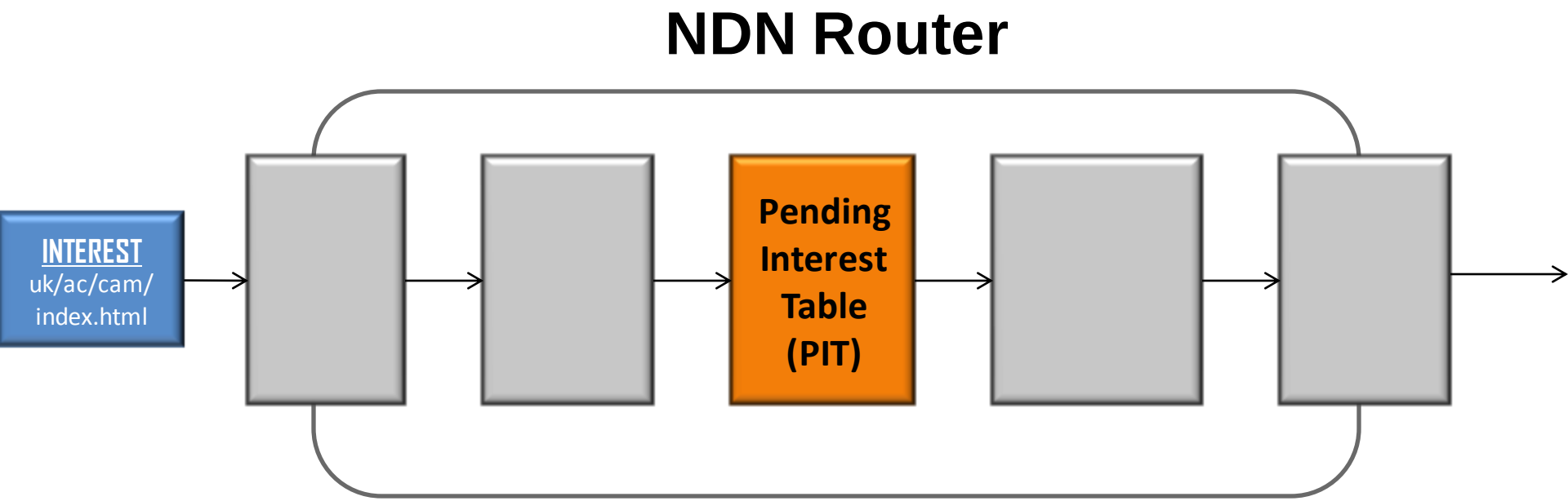


FIB hash collisions

- **PROBLEM:** Hash collisions may lead to ambiguity between entries; in some cases, bad routing decisions.

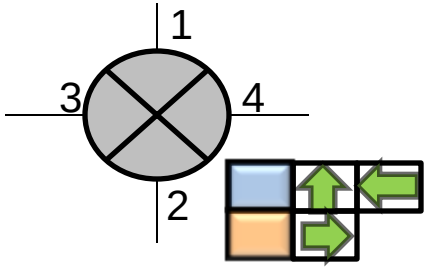
- **POSSIBLE SOLUTIONS:**
  - Wider hash outputs => higher memory requirements (double the width, double the memory ☹), but has the advantage of supporting larger namespaces;
  - Leave the problem to be solved by the control plane;
  - (FUTURE WORK) Double hashing, cuckoo hashing;

Architecture



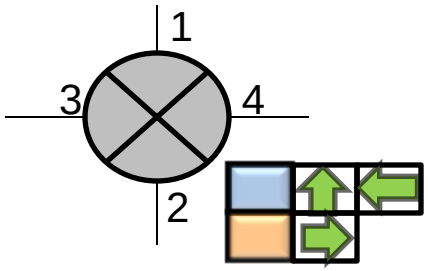
PIT

- Memorizes what each port requested.



	port			
	4	3	2	1
/uk/ac/cam/index	0	1	0	1
/pt/ul/fc/index	1	0	0	0

PIT



- Memorizes what each port requested.

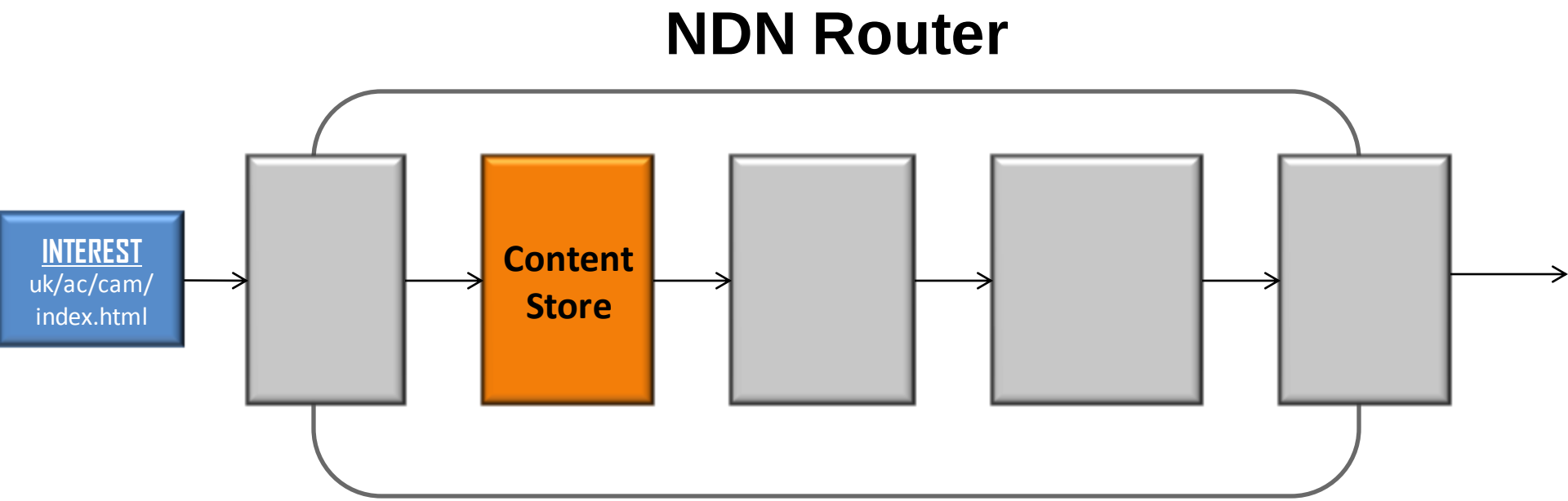
- **SOLUTION:** Implemented with 2 register arrays.
  - Each register index holds a bit vector. Each bit represents a port.

```
register<bit<MAX_PORTS>>(PIT_SIZE) PIT;
```

  - An additional register array stores hashtrays to deal with index collisions.
  - Whenever a hashtray indexes to an occupied cell, it is dropped and the consumer must reemit.

	port			
	4	3	2	1
/uk/ac/cam/index	0	1	0	1
/pt/ul/fc/index	1	0	0	0

Architecture



Content Store

- Caches packets.
  - Optional, but **key to ensuring the network's efficiency**.

- We implemented two versions in BMv2-ss**:**
  - Register implementation;
  - Directly implemented in C++ in the switch code.

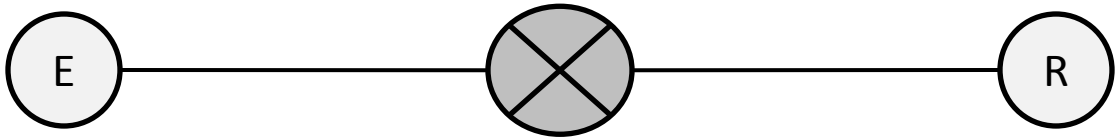
- Possible solutions when porting to hardware:
  - Register implementation;
    - Optimal performance, but competes with PIT and FIB for memory.
  - Non-volatile storage attached to the device, accessed through extern calls;
  - Port mirroring: the content store is a host connected to the device.
    - Increased latency.
    - Additional mapping required.

PART III

Evaluation

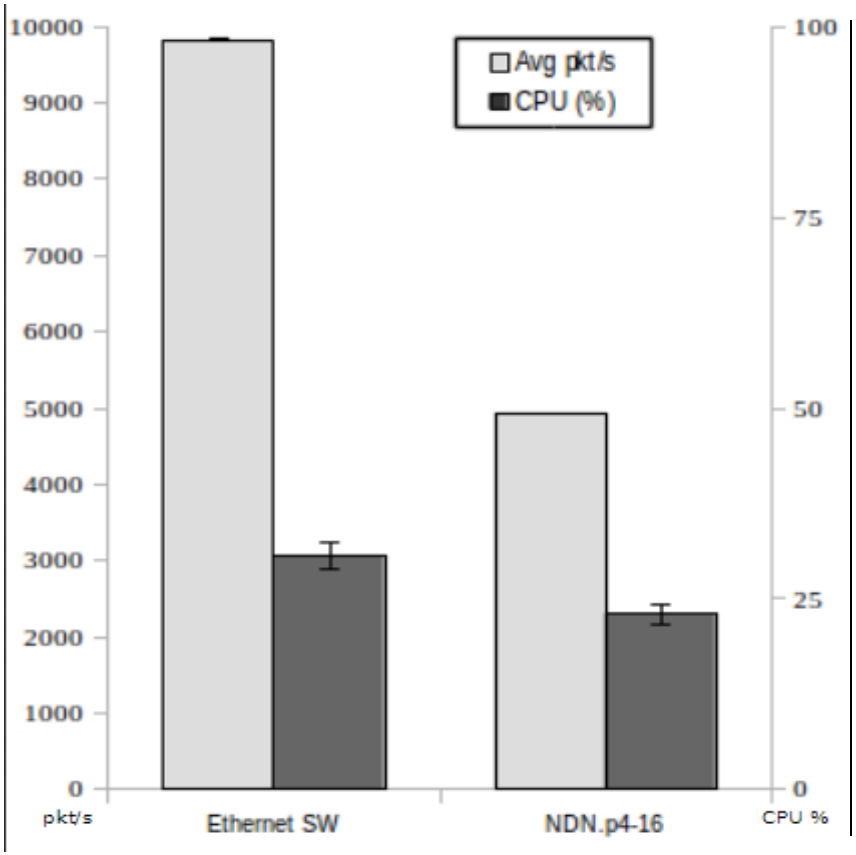
Experimental Setup

- Tests run on an off-the-shelf computer, **using Behavioral Model 2** and its **simple_switch target** (BMv2-ss). A host emits Interests, the other receives them.



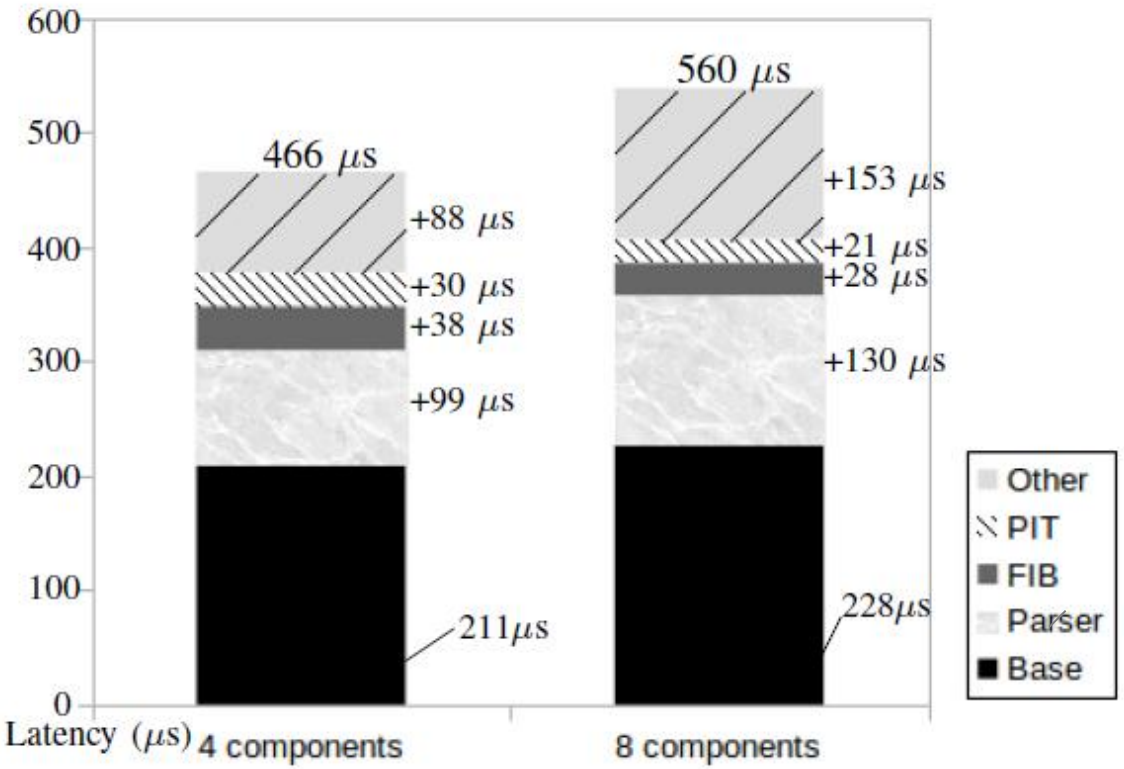
- **Two tests run:**
  - **CPU and throughput:** The emitter attempts to exhaust system resources: first with a P4 Ethernet switch (parses the Ethernet header and matches against two tables) and then with our NDN router. We collect throughput and CPU usage measures.
  - **Functional block weight:** Without exhausting the system, we find the latency of each functional block by isolating it from the others. Then, we vary the number of components in the Interest packets to assess the latency increase that results from that variation, on each of the functional blocks.

CPU and throughput



- The P4 NDN router performs many more activities than the P4 Ethernet switch, so it has less throughput.

Functional block weight



- The parser and the hashtray construction ("Other") yield higher latency as the number of components increases.
  - The other functional blocks are within the acceptable error margins

PART IV

Conclusions & Future Work

Ongoing & Future Work

- Optimize hashtray construction process, for example by parallelizing hash calculations.
- Port and adapt our solution to a hardware platform.
  - NetFPGA SUME
  - Barefoot Tofino
- Larger-scale evaluations.

Conclusions

- **NDN** is a promising new architecture **tailored for the Internet's most frequent use case: content distribution**.
  - However, **no NDN line-rate hardware exists**.
  - By consequence, **current NDN implementations are totally or partially software-based**.

- Our contribution is the design and implementation of a P4_16 NDN router that:
  - Includes all of an NDN router's functional blocks;
  - Can be ported to hardware, **and takes existing hardware in consideration** (e.g. fast TCAM for the FIB);
  - **Requires scalable amounts of memory** compared to the state-of-the-art solution.

Thank you for listening!

Questions?

Rui Miguel
(speaker)

LASIGE, Faculdade de Ciências
Universidade de Lisboa
(Faculty of Sciences, University
of Lisbon), Lisbon

fc44396@alunos.fc.ul.pt