

# Ingesting data to Hadoop

# HDFS

# Hive

# Cassandra

# HBase

These are some of the  
most popular data  
stores out there today

# Why?

HDFS

HBase

Hive

Cassandra

Why?

All of these are  
Open Source  
technologies

HDFS

HBase

Hive

Cassandra

Why?

Open Source

These are distributed  
data stores

You can scale them linearly

HDFS

HBase

Hive

Cassandra

Why?

Open Source

Scale linearly

You can use a single store  
for both analytical and  
transactional data

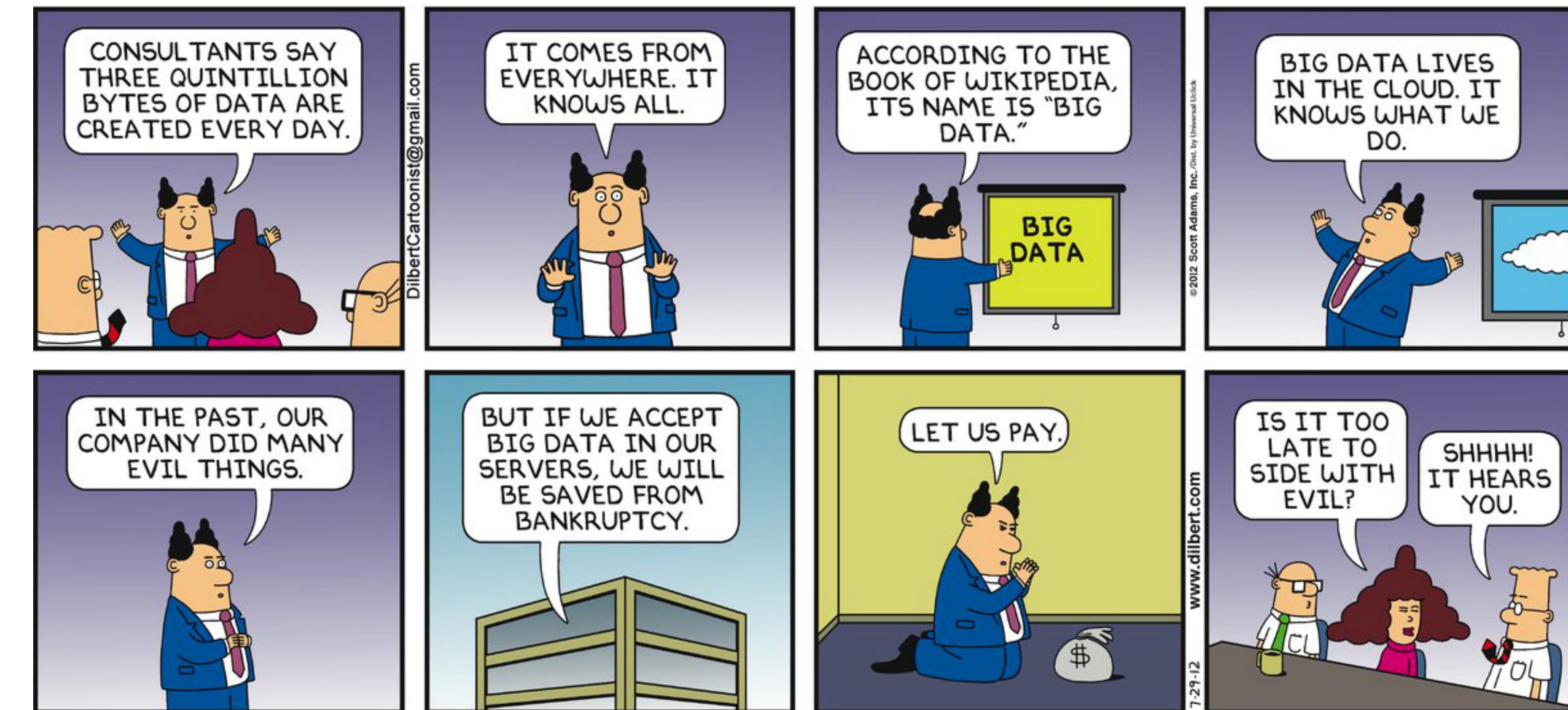
Open Source  
Scale linearly  
Analytical +  
Transactional

Whether it's for  
any of these  
perfectly  
legitimate reasons..

Whether it's for any of these  
perfectly legitimate reasons..

Open Source  
Scale linearly  
Analytical +  
Transactional

.. or because your **CIO/CTO**  
has been bitten by the  
**Big Data bug :)**



Whether it's for perfectly  
legitimate reasons..

.. or because your CIO/CTO  
has been bitten by the Big Data bug :)

HDFS  
HBase  
Hive  
Cassandra

You could be in the  
position of having to  
store your data using one  
of these technologies

Store your data using Hadoop

You've decided where  
to store your data

Store your **data** using Hadoop

Where does this  
data come from?

# Where does this data come from?

Any kind of  
application that  
produces data

Sales  
User  
notifications  
Any Web  
application

Where does this data come from?

A traditional RDBMS

This could be a legacy  
system, or an  
intermediate store before  
moving the data to Hadoop

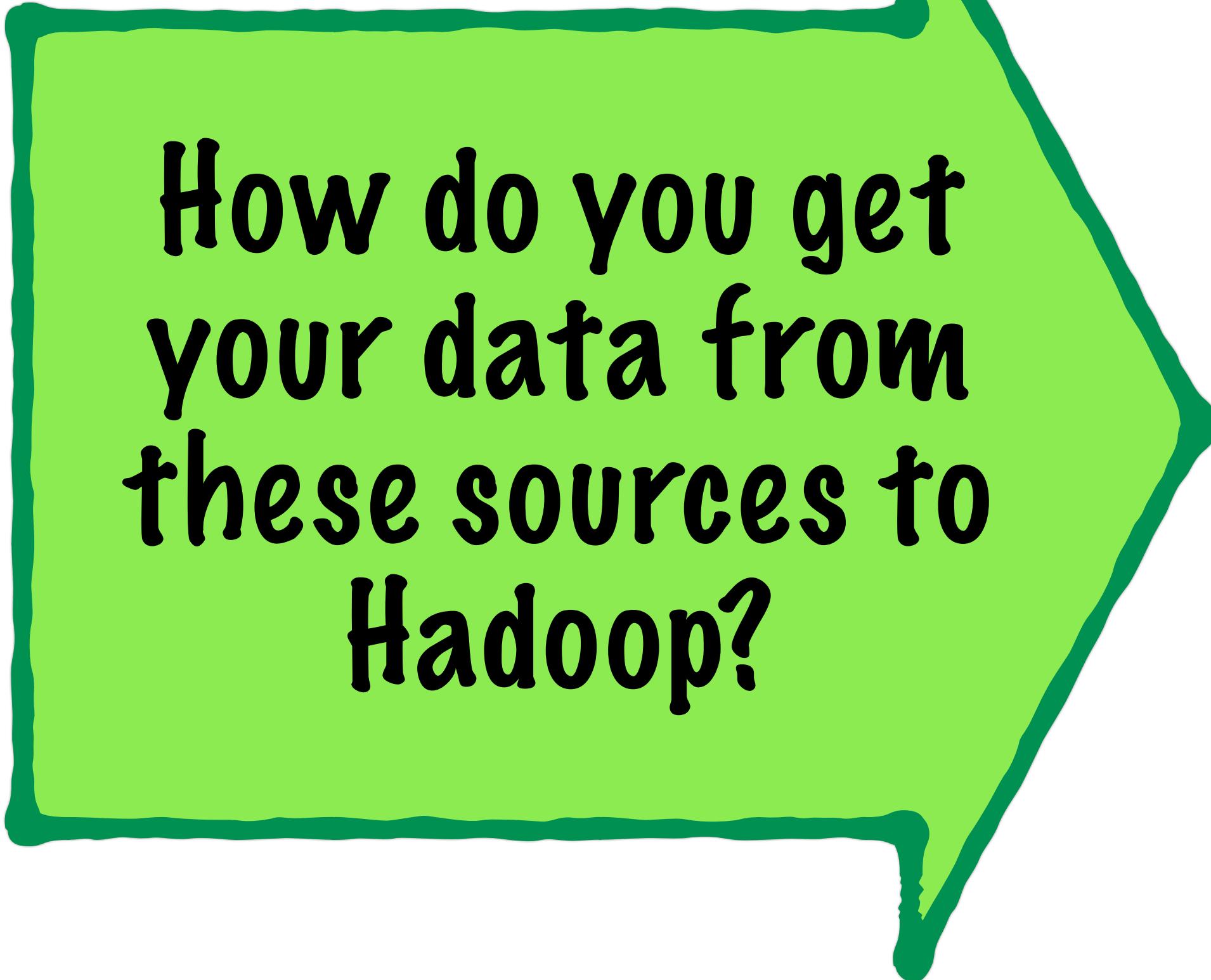
Application

MySQL

Oracle DB

SQL Server

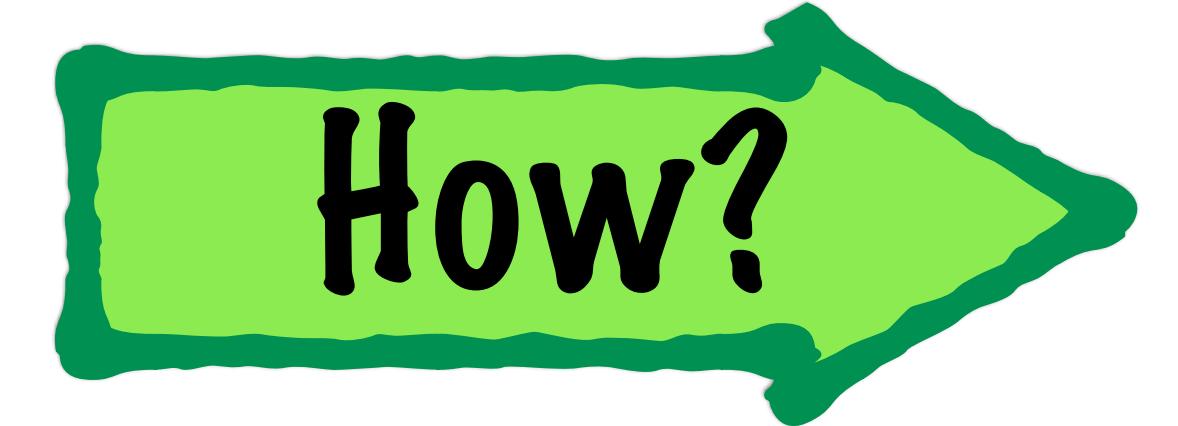
**Application  
RDBMS**



How do you get  
your data from  
these sources to  
Hadoop?

**Hadoop**

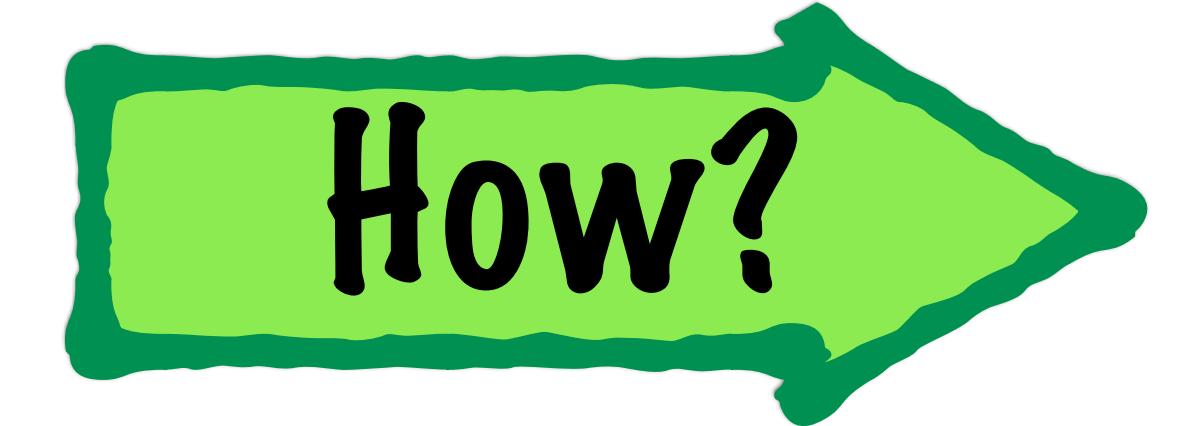
Application  
RDBMS



Hadoop

Normally, Hadoop  
ecosystem technologies  
expose Java APIs

Application  
RDBMS

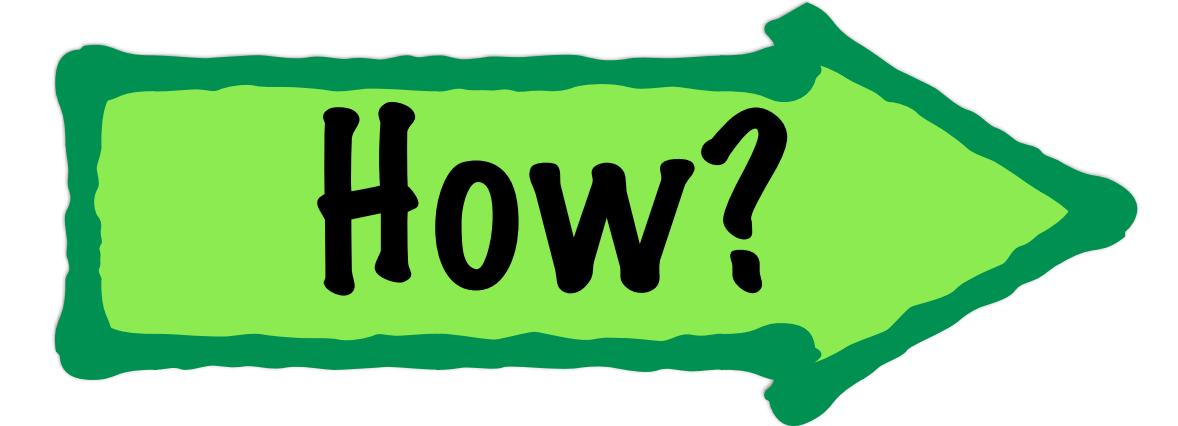


Hadoop

## Java APIs

You can directly use  
these APIs to write  
data to HDFS, HBase etc

Application  
RDBMS



Hadoop

## Java APIs

There are a few problems though, depending on the type of the source

# Application

# RDBMS

Java APIs  
There are a  
few problems

# Application

Let's say you have an application for sending and tracking user notifications

# Application user notifications

There are a number of **events** that produce data

Creating a notification

User reading a notification

User clicking on a notification

# Application

user notifications

## Events

Creating a notification

User reading a notification

User clicking on a notification

The data needs  
to be stored as  
the events occur

# Application

# user notifications

## Events

Creating a notification

User reading a notification

User clicking on a notification

This is called  
**streaming data**

# Application

Streaming data

Let's say you wanted  
to directly write  
this data to HDFS

# Application

## Streaming data to HDFS

1. Integrate your application with  
HDFS's Java API

# Application

## Streaming data to HDFS

1. Integrate your application with HDFS's Java API

2. Create a mechanism  
to buffer your data

2. Create a mechanism to  
**buffer** your data

HDFS stores data in  
the form of files

## 2. Create a mechanism to buffer your data

The HDFS files have to be large  
to take advantage of it's  
distributed architecture

## 2. Create a mechanism to buffer your data

When you write streaming data,  
you'll have to keep the HDFS file  
open until it's big enough

This is obviously a very bad idea!

## 2. Create a mechanism to **buffer** your data

Instead, buffer the data  
in-memory or in an  
intermediate file before  
writing to HDFS

# Application

Streaming data to HDFS

1. Integrate your application with HDFS's Java API
2. Create a mechanism to buffer your data

3. The **buffer layer** has  
to be fault tolerant  
and non-lossy

# Application

Streaming data to HDFS

1. Integrate your application with HDFS's Java API
2. Create a mechanism to buffer your data
3. The buffer layer has to be fault tolerant and non-lossy

Need to do all of  
this yourself

# Application

# RDBMS

Java APIs  
There are a  
few problems

# RDBMS

Let's say you have a  
legacy system that  
used an RDBMS

# RDBMS

To port your data  
to HDFS

## Option 1

1. Dump your tables  
into large files

2. Manually copy  
these files to HDFS

# RDBMS

## To port your data to HDFS

Option 1 Dump and copy

Option 2 Use scripts to

Read the data from RDBMS

Write to HDFS

# RDBMS

# To port your data to HDFS

Option 1 Dump and copy

Option 2 Use scripts

Either of these is  
pretty onerous to do

# Application

# RDBMS

Java APIs  
There are a  
few problems

Application  
RDBMS

Java APIs  
There are a  
few problems

## Flume and Sqoop

are technologies developed to isolate  
and abstract the transport of data  
between a source and data store

# Flume and Sqoop

are both open source technologies

developed and maintained by

# THE APACHE SOFTWARE FOUNDATION

as part of the Hadoop ecosystem

# Flume

Flume acts as a **buffer** between  
your streaming application and  
HDFS/other data stores

Application

Flume

Data Store

Flume can

Read data from  
many types of  
sources

HTTP, a file  
directory, Syslog  
messages

# Application

## Flume

# Data Store

Read

Write data to  
many types of  
sinks

HDFS, HBase,  
Local files etc

# Application

# Flume

# Data Store

Read  
Write

Buffer the data based on  
the requirements of the  
source and the sink

# Application

Flume

# Data Store

Read  
Write

## Buffer

Sources might produce data at different rates and in different formats

# Application

## Flume

# Data Store

Read  
Write  
**Buffer**

**Sinks** might require data to be  
written in a particular  
format at a particular rate

# Application



Flume

# Data Store

Read  
Write  
**Buffer**

For many commonly used sources and sinks, Flume comes with built-in handlers to do all of this

# Application

## Flume

# Data Store

Read  
Write  
**Buffer**

Flume also takes care of fault tolerance and guarantees no data loss for certain configurations

# Application



Flume

# Data Store

Flume is a push based system  
ie. it decides when the data  
should be written to the sink

# Sqoop

Sqoop is a tool that can  
directly import data  
from RDBMS to Hadoop

# Sqoop

Unlike Flume, Sqoop is

Pull based

Used for bulk-imports (not  
streaming data)

# Sqoop

Sqoop comes with connectors  
for many popular RDBMS

MySQL, Oracle, SQL  
Server, PostgreSQL etc

# Sqoop

With one command line  
action, import data from  
RDBMS to HDFS/Hive

# Sqoop

You can import entire  
tables, or the results of  
specified SQL queries

# Sqoop

You can also schedule  
periodic imports using  
Sqoop jobs

# Flume and Sqoop

integrate really well  
with the data stores in  
the Hadoop ecosystem

# Flume

Let's go through a few  
different use cases for  
**Flume**

# Flume Agent

# Flume Agent

A Flume Agent is the  
simplest unit of Flume

# Flume Agent

A Flume Agent can connect  
any number of sources to  
any number of data stores

# Flume Agent

A Flume Agent represents 1 hop for the data



# Flume Agent

If the data has to go through **multiple hops**, use a chain of Flume Agents



# Flume Agent

Every Flume Agent has 3 basic components

A component  
to receive  
data

A component  
to buffer  
data

A component  
to write  
data

# Flume Agent



# Flume Agent



The Source component  
receives data and pushes it  
to the channel

# Flume Agent



The Source component **listens to the specified source and understands the format in which data is being sent**

# Flume Agent

Source

Channel

Sink

Flume supports many types of Sources

## Spooling Directory

A directory to which files are written by an application

# Flume Agent



**Spooling Directory**

## Syslog

Log messages written in the  
syslog format

# Flume Agent

Source

Channel

Sink

Spooling Directory  
Syslog

Exec

Standard output from any  
command

# Flume Agent

Source

Channel

Sink

Spooling Directory

Syslog

Exec

## Custom Sources

Write your own sources which can receive and understand the format of data produced by your application

# Flume Agent



Spooling Directory  
Syslog  
Exec  
Custom Sources

1 Flume Agent can  
have multiple  
Source components

# Flume Agent



Sniffing Directory  
Filebeat Adapter  
Logstash Adapter

Sources receive  
data and push it to  
a **Channel**

# Flume Agent

Source

Channel

Sink

The Channel is a **buffer** where data stays **until** the **Sink writes** it out to an endpoint

# Flume Agent

Source

Channel

Sink

The buffer capacity  
can be configured by  
the user

# Flume Agent



It's important to configure  
the Channel capacity as  
per the rates of the  
Source and the Sink

# Flume Agent



Streaming Directory

If the Channel gets filled up  
before the Sink is able to read  
from it, it will stop accepting  
more data from the Source

# Flume Agent

Source

Channel

Sink

There are many types  
of channels, but 2 are  
commonly used

# Flume Agent



## Memory

Memory channels buffer  
the data in-memory

# Flume Agent



## Memory

This channel acts like  
an **in-memory** queue

# Flume Agent



## Memory

Sources write to the  
tail of the queue

# Flume Agent



## Memory

Sinks read from the  
head of the queue

# Flume Agent



Memory

File

The memory channel is not  
persistent

# Flume Agent



Streaming Directory  
Memory  
**File**

If there is a crash/restart,  
the data in the channel is lost

# Flume Agent



Spooling Directory  
Base directory

# File

## Memory

File channels are better when you  
want a guarantee of no data loss

# Flume Agent



Spooling Directory  
Memory

## File

Since the data is buffered on disk,  
you can also have a much larger  
buffer capacity

# Flume Agent

Source

Channel

Sink

Memory  
File

Channels are continuously  
polled by Sinks, which write  
the data to the endpoint

# Flume Agent



Spooling Directory  
Syslog  
Exec  
Custom Sources

Memory File

Sink

1 Sink can only  
read from 1 channel

# Flume Agent

Source

Channel

Sink

Spooling Directory

Syslog

Exec

Custom Sources

Memory  
File

The Sink will keep  
polling the Channel to  
read and remove data

# Flume Agent

Source

Channel

Sink

Spooling Directory

Syslog

Exec

Custom Source

Memory

FIFO

Once the data is **safely written** to the end point,  
the Sink will inform the Channel to delete it

# Flume Agent

Flume supports many  
different Sinks

Sink

Console Log

HDFS

Local Directory

HBase

# Flume Agent



It's the built-in support for  
HDFS and HBase that  
makes Flume a good choice  
for Hadoop ecosystem users

HDFS  
HBase

# Flume Agent

Source

Spooling Directory  
Syslog  
Exec  
Custom Sources

Channel

Memory  
File

Sink

HDFS  
HBase  
Console Log  
Local  
Directory

**Example 1:**

**Spool to Logger**

# Example 1: Spool to Logger

Say you have an application  
that generates files and writes  
them to a specified directory

Ex: Logfiles

# Example 1: Spool to Logger

You can configure Flume to  
“watch” for new files in the  
directory

This is called a Spooling  
Directory

# Example 1: Spool to Logger

## Spooling Directory

Whenever a new file is added to the directory

1. Flume will read it
2. Eventually push the data to a specified endpoint

# Example 1: Spool to Logger

Spooling  
Directory

Console log

In this case, we are simply pushing the data to a console log

Spooling  
Directory

Flume  
Agent

Console log

In between these 2, we need  
to set up a Flume Agent

# Example 1: Spool to Logger



To set up a Flume Agent, configure all the options in a **.properties** file

# Example 1: Spool to Logger

Spooling  
Directory



Console log

Start the Flume agent from  
the command line

# Example 1: Spool to Logger

Spooling  
Directory



Console log

```
$ flume-ng agent  
--conf-file spool-to-logger.properties  
--name agent1
```

# Example 1: Spool to Logger

```
$ flume-ng agent  
--conf-file spool-to-logger.properties  
--name agent1
```

You can only start 1  
Flume agent at a time

# Example 1: Spool to Logger

```
$ flume-ng agent  
--conf-file spool-to-logger.properties  
--name agent1
```

The command to  
start a Flume agent

# Example 1: Spool to Logger

\$ **flume-ng agent**

--conf-file spool-to-logger.properties

--name agent1

**flume-ng** stands for

**flume next generation**

This is a later version of **Flume**

# Example 1: Spool to Logger

```
$ flume-ng agent  
--conf-file spool-to-logger.properties  
--name agent1
```

Use the name option to  
specify the **name of the**  
**agent**

# Example 1: Spool to Logger

```
$ flume-ng agent  
--conf-file spool-to-logger.properties  
--name agent1
```

Flume will look in the properties file for the configuration of this agent

# Example 1: Spool to Logger

```
$ flume-ng agent  
--conf-file spool-to-logger.properties  
--name agent1
```

You can configure **multiple agents** in the same properties file

# Example 1: Spool to Logger

```
$ flume-ng agent  
--conf-file spool-to-logger.properties  
--name agent1
```

Flume will only pick up the configuration for the specified agent name

# Example 1: Spool to Logger

```
$ flume-ng agent  
--conf-file spool-to-logger.properties  
--name agent1
```

Specify the path to  
the properties file

## spool-to-logger.properties

```
agent1.sources =source1  
agent1.sinks = sink1  
agent1.channels = channel1
```

Properties are specified  
in a hierarchical  
manner

```
agent1.sources.source1.channels = channel1  
agent1.sinks.sink1.channels = channel1  
agent1.sources.source1.type = spooldir  
agent1.sources.source1.spoolDir = /Users/swethakolalapudi/tmp/spooldir  
agent1.sinks.sink1.type = logger  
agent1.channels.channel1.type = file
```

# spool-to-logger.properties

```
agent1.  
agent1.  
agent1.
```

```
agent1.  
agent1.
```

```
agent1.  
agent1.
```

```
agent1.  
agent1.
```

At the top level, you  
have the name of the  
**flume agent**

```
$ flume-ng agent  
--conf-file spool-to-logger.prope  
--name agent1
```

## **spool-to-logger.properties**

The agent has **sources**,  
**sinks** and **channels**

```
|agent1.sources =source1  
agent1.sinks = sink1  
agent1.channels = channel1
```

## **spool-to-logger.properties**

If there are **multiple**  
**sources/channels/sinks** -  
separate them by a space

```
|agent1.sources =source1  
agent1.sinks = sink1  
agent1.channels = channel1
```

## spool-to-logger.properties

source1

channel1

sink1

```
agent1.sources =source1  
agent1.sinks = sink1  
agent1.channels = channel1
```

# spool-to-logger.properties



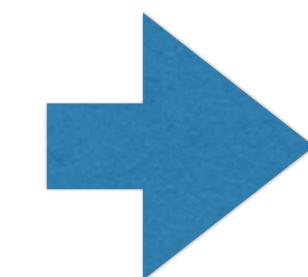
```
agent1.sources.source1.channels = channel1
```

Each source has  
channels

## spool-to-logger.properties

source1

channel1



sink1

```
agent1.sinks.sink1.channel = channel1
```

Each sink also has a  
channel

## spool-to-logger.properties



```
agent1.sources.source1.channels = channel1  
agent1.sinks.sink1.channel = channel1
```

Note that a source can have multiple channels, but a sink can have only 1 channel

## spool-to-logger.properties

source1

channel1

sink1

```
agent1.sources.source1.type = spooldir
```

```
agent1.sources.source1.spoolDir  = /Users/swethakolalapudi/tmp/spooldir
```

The source type is  
**spooldir**

## spool-to-logger.properties

source1

channel1

sink1

```
agent1.sources.source1.type = spooldir
```

```
agent1.sources.source1.spoolDir  = /Users/swethakolalapudi/tmp/spooldir
```

We need to specify the path  
of the Spooling directory

## spool-to-logger.properties

source1

channel1

sink1

```
agent1.sinks.sink1.type = logger
```

The sink type is logger

# spool-to-logger.properties

source1

channel1

sink1

```
agent1.channels.channel1.type = file
```

The channel type is file

# spool-to-logger.properties

```
agent1.sources =source1
```

```
agent1.sinks = sink1
```

```
agent1.channels = channel1
```

```
agent1.sources.source1.channels = channel1
```

```
agent1.sinks.sink1.channel = channel1
```

```
agent1.sources.source1.type = spooldir
```

```
agent1.sources.source1.spoolDir = /Users/swethakolalapudi/tmp/spo
```

```
agent1.sinks.sink1.type = logger
```

```
agent1.channels.channel1.type = file
```

# Example 1: Spool to Logger

```
$ flume-ng agent  
--conf-file spool-to-logger.properties  
--name agent1
```

Now whenever you add a file to the Spooling directory, its contents will appear on the console log

# Example 1: Spool to Logger

A couple of important notes

Files added to the Spooling Directory should be

- # 1: Text files
- # 2: Immutable
- # 3: Unique in name

# Example 1: Spool to Logger

# 1: Text files

The Spooling Directory  
Source expects that the files  
written will be text files

# Example 1: Spool to Logger

## # 1: Text files

To read files in any other format, you have to write a **custom deserializer in Java** and plug it in your properties file

# Example 1: Spool to Logger

# 1: Text files  
# 2: Immutable

Once Flume reads a file,  
it will mark it as  
completed and move on

# Example 1: Spool to Logger

# 1: Text files

# 2: Immutable

Flume expects that  
you will not try to  
modify the file later

# Example 1: Spool to Logger

# 1: Text files

# 2: Immutable

# 3: Unique in name

If you try to add a file with  
the same name as a previous  
one, the Flume Agent will  
throw an exception and quit

# Example 1: Spool to Logger

- # 1: Text files
- # 2: Immutable
- # 3: Unique in name

Things to keep in mind  
when using a Spooling  
Directory Source

# Flume Events

# Flume Events



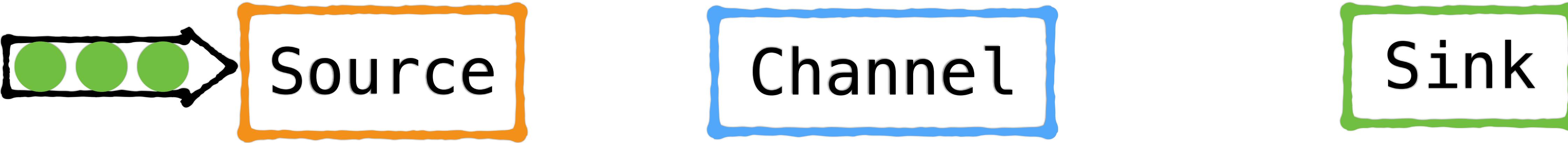
We've spoken about how  
the Flume Agent has  
Source, Channels and  
Sinks to read/write data

# Flume Events



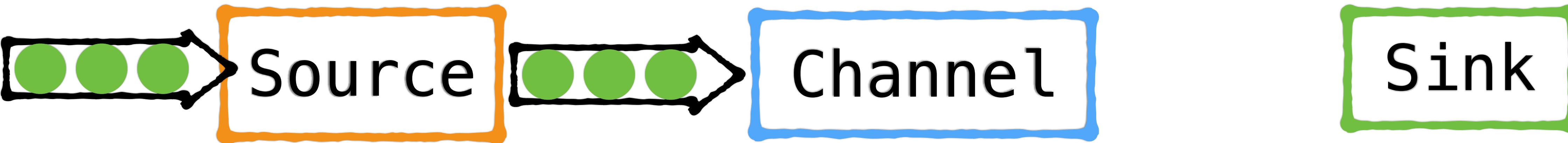
A **Flume Event** is the base  
unit of communication  
between these components

# Flume Events



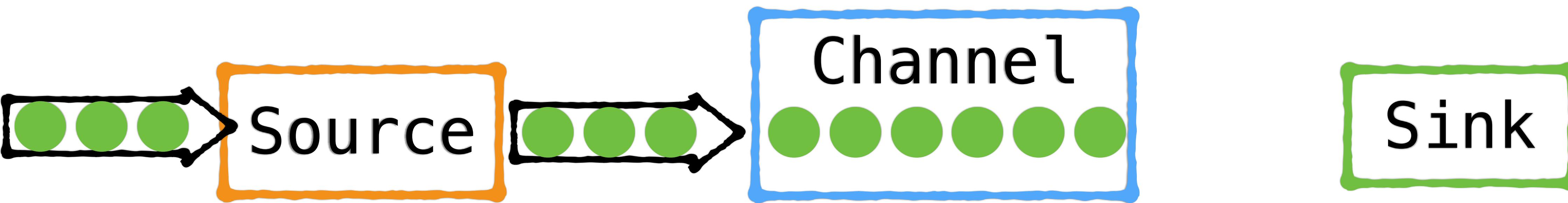
Sources receive data  
in the form of events

# Flume Events



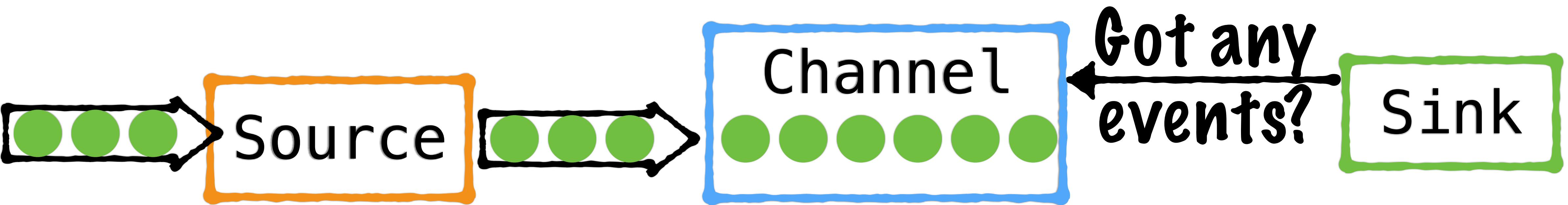
Sources push events to  
the channel

# Flume Events



Channels buffer  
events

# Flume Events



Sinks poll the channel  
for events

# Flume Events



The Sink reads the events  
and forwards them to the  
end point

# Flume Events

An event is a **key value pair**  
representing 1 record of data

Key = Event Headers

Value = Event Body

# Flume Events

Event Headers

Event Body

All data is transported  
by Flume in the form of  
events

# Flume Events

Event Headers

Event Body

For instance: with a Spooling  
Directory source, each new  
file becomes a new event

# Flume Events

Event Headers

Event Body

The event body is the  
actual payload/data to  
be transported

# Flume Events

Event Headers

Event Body

This is usually a byte  
array representing a  
single record of data

# Flume Events

Event Headers

Event Body

The event body has to  
fit in the channel

# Flume Events

Event Headers

Event Body

If the event body exceeds the channel capacity, then Flume won't be able to transport that event

# Flume Events

Event Headers

Event Body

For instance, if you want to transport very large files to HDFS, then a direct copy might be a better choice than using Flume

# Flume Events

Event Headers

Event Body

Event Headers  
contain meta data

# Flume Events

## Event Headers

Event Body

Flume might use this meta  
data to route events  
between sources and  
channels

# Flume Events

## Event Headers

Event Body

Headers can also be  
used to carry event  
IDs or Uuids for events

**Example 2:**

**Spool to HDFS**

# Example 2: Spool to HDFS

We've seen how to read data from a Spooling Directory Source and push it to the console log

## Example 2: Spool to **HDFS**

Spooling  
Directory



This time, let's set up the agent to push the data to HDFS

## Example 2: Spool to **HDFS**

Spooling  
Directory



Edit the properties file to  
change the sink to **HDFS**

## spool-to-logger.properties

```
agent1.sources =source1  
agent1.sinks = sink1  
agent1.channels = channel1
```

```
agent1.sources.source1.channels = channel1  
agent1.sinks.sink1.channel = channel1
```

```
agent1.sources.source1.type = spooldir  
agent1.sources.source1.spoolDir = /Users/swethakolalapudi/tmp/spooldir
```

```
agent1.sinks.sink1.type = logger  
agent1.channels.channel1.type = file
```

# The old properties file

## spool-to-logger.properties

```
agent1.sources =source1  
agent1.sinks = sink1  
agent1.channels = channel1
```

```
agent1.sources.source1.channels = channel1  
agent1.sinks.sink1.channel = channel1
```

```
agent1.sources.source1.type = spooldir  
agent1.sources.source1.spoolDir = /Users/swethakolalapudi/tmp/spooldir
```

```
agent1.sinks.sink1.type = logger  
agent1.channels.channel1.type = file
```

The sink type is logger

## spool-to-hdfs.properties

```
agent1.sinks.sink1.type = hdfs
agent1.sinks.sink1.hdfs.path = /tmp/flume
agent1.sinks.sink1.hdfs.filePrefix = events
agent1.sinks.sink1.hdfs.fileSuffix = .log
agent1.sinks.sink1.hdfs.inUsePrefix =
agent1.sinks.sink1.hdfs fileType = DataStream
```

The HDFS sink needs some additional properties to be configured

## spool-to-hdfs.properties

```
agent1.sinks.sink1.type = hdfs  
agent1.sinks.sink1.hdfs.path = /tmp/flume  
agent1.sinks.sink1.hdfs.filePrefix = events  
agent1.sinks.sink1.hdfs.fileSuffix = .log  
agent1.sinks.sink1.hdfs.inUsePrefix = _  
agent1.sinks.sink1.hdfs.fileType = DataStream
```

The sink type is set to hdfs

## spool-to-hdfs.properties

```
agent1.sinks.sink1.type = hdfs  
agent1.sinks.sink1.hdfs.path = /tmp/flume  
agent1.sinks.sink1.hdfs.filePrefix = events  
agent1.sinks.sink1.hdfs.fileSuffix = .log  
agent1.sinks.sink1.hdfs.inUsePrefix = _  
agent1.sinks.sink1.hdfs fileType = DataStream
```

hdfs is the alias for a HDFS  
sink

## spool-to-hdfs.properties

```
agent1.sinks.sink1.type = hdfs  
agent1.sinks.sink1.hdfs.path = /tmp/flume  
agent1.sinks.sink1.hdfs.filePrefix = events  
agent1.sinks.sink1.hdfs.fileSuffix = .log  
agent1.sinks.sink1.hdfs.inUsePrefix = _  
agent1.sinks.sink1.hdfs fileType = DataStream
```

You can also use the **fully qualified class name** for the sink

**org.apache.flume.sink.hdfs.HDFSEventSink**

## spool-to-hdfs.properties

```
agent1.sinks.sink1.type = hdfs  
agent1.sinks.sink1.hdfs.path = /tmp/flume  
agent1.sinks.sink1.hdfs.filePrefix = events  
agent1.sinks.sink1.hdfs.fileSuffix = .log  
agent1.sinks.sink1.hdfs.inUsePrefix = _  
agent1.sinks.sink1.hdfs.fileType = DataStream
```

The path to which the data will be written

## spool-to-hdfs.properties

```
agent1.sinks.sink1.hdfs.filePrefix = events  
agent1.sinks.sink1.hdfs.fileSuffix = .log
```

A file prefix and suffix for the files that will be written

## spool-to-hdfs.properties

```
agent1.sinks.sink1.hdfs.filePrefix = events  
agent1.sinks.sink1.hdfs.fileSuffix = .log
```

Filenames will be  
*events\_<some number>.log*

## spool-to-hdfs.properties

```
agent1.sinks.sink1.hdfs.filePrefix = events  
agent1.sinks.sink1.hdfs.fileSuffix = .log
```

Note that the file names of files in  
the Spooling directory **are not**  
**preserved**

## spool-to-hdfs.properties

```
agent1.sinks.sink1.hdfs.filePrefix = events  
agent1.sinks.sink1.hdfs.fileSuffix = .log
```

The contents of each file in the Spooling directory will become the body of 1 event

## spool-to-hdfs.properties

```
agent1.sinks.sink1.hdfs.filePrefix = events  
agent1.sinks.sink1.hdfs.fileSuffix = .log
```

These events are written  
later to **HDFS files**

## spool-to-hdfs.properties

```
agent1.sinks.sink1.hdfs.filePrefix = events  
agent1.sinks.sink1.hdfs.fileSuffix = .log
```

The files in HDFS are rolled  
over every 30 seconds

## spool-to-hdfs.properties

```
agent1.sinks.sink1.hdfs.filePrefix = events  
agent1.sinks.sink1.hdfs.fileSuffix = .log
```

You can change the interval by  
setting the **rollInterval**  
property

## spool-to-hdfs.properties

```
agent1.sinks.sink1.hdfs.filePrefix = events  
agent1.sinks.sink1.hdfs.fileSuffix = .log
```

Instead of rolling over by time, you can roll over the files **by event count or cumulative event size**

# spool-to-hdfs.properties

```
agent1.sinks.sink1.hdfs.filePrefix = events  
agent1.sinks.sink1.hdfs.fileSuffix = .log
```

event count

rollCount

cumulative event size

rollSize

## spool-to-hdfs.properties

```
agent1.sinks.sink1.hdfs.inUsePrefix = _
```

Prefix used when a file is  
open and being written to

## spool-to-hdfs.properties

```
agent1.sinks.sink1.hdfs.fileType = DataStream
```

The file format of the  
**HD**FS** files**

## spool-to-hdfs.properties

```
agent1.sinks.sink1.hdfs.fileType = DataStream
```

The default option is  
SequenceFile

## spool-to-hdfs.properties

```
agent1.sinks.sink1.hdfs.fileType = DataStream
```

SequenceFile

is used when you want to write  
the events in binary format

## spool-to-hdfs.properties

```
agent1.sinks.sink1.hdfs.fileType = DataStream
```

## SequenceFile

It's expected that whoever is reading the files knows how to serialize the binary data to objects/data

## spool-to-hdfs.properties

```
agent1.sinks.sink1.hdfs.fileType = DataStream
```

**DataStream**

For uncompressed  
file formats

**CompressedStream**

For compressed file  
formats (gzip,bzip2  
etc)

## spool-to-hdfs.properties

```
agent1.sinks.sink1.hdfs.fileType = DataStream
```

**DataStream**  
For uncompressed  
file formats

The default file  
format is **TEXT**

## spool-to-hdfs.properties

```
agent1.sinks.sink1.hdfs.fileType = DataStream
```

DataStream  
For uncompressed  
file formats

For other formats, you have to  
**write a serializer class** and plug  
it in the sink's serializer property

## spool-to-hdfs.properties

```
agent1.sinks.sink1.type = hdfs  
agent1.sinks.sink1.hdfs.path = /tmp/flume  
agent1.sinks.sink1.hdfs.filePrefix = events  
agent1.sinks.sink1.hdfs.fileSuffix = .log  
agent1.sinks.sink1.hdfs.inUsePrefix = _  
agent1.sinks.sink1.hdfs fileType = DataStream
```

This sets up the HDFS sink  
for your Flume Agent!

**Example 3:**

**HTTP to HDFS**

## Example 3: HTTP to HDFS

Let's say you are tracking  
clicks on a webpage

# Example 3: HTTP to HDFS

## tracking clicks

Every click might generate  
some actions in the form of  
**HTTP requests**

# Example 3: HTTP to HDFS tracking clicks

One of these HTTP requests  
could be posting an event to  
Flume

# Example 3: HTTP to HDFS

HTTP Source

HTTP Source actually sets up a  
webserver at a specified host  
and port

# Example 3: HTTP to HDFS

## HTTP Source

HTTP Source acts just as if there were a webserver at a specified host and port

# Example 3: HTTP to HDFS

## HTTP Source

The requests should be posted in the form of a JSON

```
{ "headers": {},  
  "body": }
```

# Example 3: HTTP to HDFS

HTTP Source

You can also write **custom  
HTTP handlers** for events  
posted in other formats

# Example 3: HTTP to HDFS

```
httpagent.sources = http-source
httpagent.sinks = hdfs-sink
httpagent.channels = ch

httpagent.sources.http-source.channels = ch
httpagent.sinks.hdfs-sink.channel = ch

# Define / Configure Source
#####
httpagent.sources.http-source.type = org.apache.flume.source.http.HTTPSource
httpagent.sources.http-source.channels = ch
httpagent.sources.http-source.bind = localhost
httpagent.sources.http-source.port = 8989
```

```
# HDFS File Sink
#####
httpagent.sinks.hdfs-sink.t
httpagent.sinks.hdfs-sink.h
httpagent.sinks.hdfs-sink.h
httpagent.sinks.hdfs-sink.h
httpagent.sinks.hdfs-sink.h
httpagent.sinks.hdfs-sink.h
```

# http-to-hdfs.properties

```
# Channels
#####
httpagent.channels.ch.type = memory
httpagent.channels.ch.capacity = 1000
```

# Example 3: HTTP to HDFS

```
httpagent.sources = http-source  
httpagent.sinks = hdfs-sink  
httpagent.channels = ch
```

httpagent is the name of  
the Flume agent

# Example 3: HTTP to HDFS

```
httpagent.sources = http-source  
httpagent.sinks = hdfs-sink  
httpagent.channels = ch
```

It has a source, channel and sink

http-source

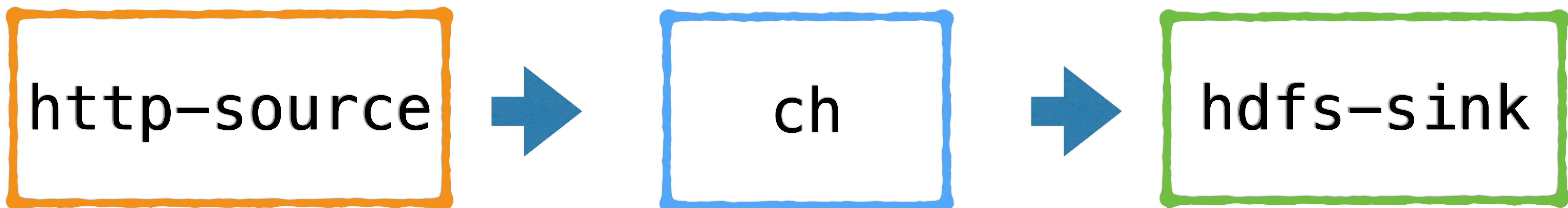
ch

hdfs-sink

# Example 3: HTTP to HDFS

```
httpagent.sources.http-source.channels = ch  
httpagent.sinks.hdfs-sink.channel = ch
```

Connect the source and sink to the channel



# Example 3: HTTP to HDFS

hdfs-sink

```
# HDFS File Sink
#####
httpagent.sinks.hdfs-sink.type = hdfs
httpagent.sinks.hdfs-sink.hdfs.path = /tmp/flume/hdfs
httpagent.sinks.hdfs-sink.hdfs.filePrefix = events
httpagent.sinks.hdfs-sink.hdfs.fileSuffix = .log
httpagent.sinks.hdfs-sink.hdfs.inUsePrefix =
httpagent.sinks.hdfs-sink.hdfs.fileType = DataStream
```

Set up the HDFS sink as usual

# Example 3: HTTP to HDFS

ch

```
# Channels
#####
httpagent.channels.ch.type = memory
httpagent.channels.ch.capacity = 1000
```

We are using a **memory channel**  
instead of a file channel

# Example 3: HTTP to HDFS

ch

```
# Channels  
#####  
httpagent.channels.ch.type = memory  
httpagent.channels.ch.capacity = 1000
```

Capacity is the number of events the channel can hold

# Example 3: HTTP to HDFS

ch

```
# Channels
#####
httpagent.channels.ch.type = memory
httpagent.channels.ch.capacity = 1000
```

If the number of events unread by the sink goes over 1000, the channel will stop accepting more events

# Example 3: HTTP to HDFS

http-source

```
httpagent.sources.http-source.type = org.apache.flume.source.http.HTTPSource  
httpagent.sources.http-source.channels = ch  
httpagent.sources.http-source.bind = localhost  
httpagent.sources.http-source.port = 8989
```

# The source configuration

# Example 3: HTTP to HDFS

http-source

```
httpagent.sources.http-source.type = org.apache.flume.source.http.HTTPSource
```

The source type is set to  
**HTTPSource**

# Example 3: HTTP to HDFS

http-source

```
httpagent.sources.http-source.bind = localhost  
httpagent.sources.http-source.port = 8989
```

The host and port where the HTTP POST requests should be sent

# Example 3: HTTP to HDFS

To test this, first  
start the Flume agent

```
$ flume-ng agent  
--conf-file http-to-hdfs.properties  
--name httpagent
```

# Example 3: HTTP to HDFS

From another terminal,  
send a POST request

```
$ curl -X POST \
-H 'Content-Type: application/json; charset=UTF-8' \
-d '[ {"headers" : {"eventheader1" : "event1", \
"eventheader2" : "event2" } , \
"body" : "This is the body1" }]' \
http://localhost:8989
```

# Example 3: HTTP to HDFS

A POST request to  
the specified host

```
$ curl -X POST \
-H 'Content-Type: application/json; charset=UTF-8' \
-d '[ {"headers" : {"eventheader1" : "event1", \
"eventheader2" : "event2" } , \
"body" : "This is the body1" }]' \
http://localhost:8989
```

# Example 3: HTTP to HDFS

The format is JSON in UTF-8  
(can also use UTF-16/UTF-32)

```
$ curl -X POST \
-H 'Content-Type: application/json; charset=UTF-8' \
-d '[ {"headers" : {"eventheader1" : "event1", \
"eventheader2" : "event2" } , \
"body" : "This is the body1" }]' \
http://localhost:8989
```

# Example 3: HTTP to HDFS

## The JSON representing the event

```
$ curl -X POST \  
-H 'Content-Type: application/json; charset=UTF-8' \  
-d '[ {"headers" : {"eventheader1" : "event1", \  
"eventheader2" : "event2" } , \  
"body" : "This is the body1" }]' \  
http://localhost:8989
```

# Example 3: HTTP to HDFS

## The headers / metadata

```
$ curl -X POST \  
-H 'Content-Type: application/json; charset=UTF-8' \  
-d '[ {"headers" : {"eventheader1" : "event1", \  
"eventheader2" : "event2" } , \  
"body" : "This is the body1" }]' \  
http://localhost:8989
```

# Example 3: HTTP to HDFS

The body/actual payload  
to be transported

```
$ curl -X POST \
-H 'Content-Type: application/json; charset=UTF-8' \
-d ' [ {"headers" : {"eventheader1" : "event1", \
"eventheader2" : "event2" } , \
"body" : "This is the body1" }] ' \
http://localhost:8989
```

# Example 4:

HTTP to HDFS with  
*event bucketing*

# Example 4: event bucketing

Let's go back to our **HTTP to HDFS** example

We saw how to POST events  
using a **JSON** format

```
$ curl -X POST \
-H 'Content-Type: application/json; charset=UTF-8' \
-d '[ {"headers" : {"eventheader1" : "event1", \
"eventheader2" : "event2" } , \
"body" : "This is the body1" }]' \
http://localhost:8989
```

# Example 4: event bucketing

## JSON format

```
{  
  "headers" : {  
    "eventheader1" : "event1",  
    "eventheader2" : "event2"  
  },  
  "body" : "This is the body1"  
}
```

## Example 4: event bucketing

```
{  
  "headers" : {  
    "eventheader1" : "event1",  
    "eventheader2" : "event2"  
  },  
  "body" : "This is the body1"  
}
```

This is the event body

## Example 4: event bucketing

```
{  
  "headers" : {  
    "eventheader1" : "event1",  
    "eventheader2" : "event2"  
  },  
  "body" : "This is the body1"  
}
```

This is the data we actually  
see in the HDFS files

## Example 4: event bucketing

```
{  
  "headers" : {  
    "eventheader1" : "event1",  
    "eventheader2" : "event2"  
  },  
  "body" : "This is the body1"  
}
```

This is metadata that Flume can use  
to decide how to deal with events

## Example 4: event bucketing

```
{  
  "headers" : {  
    "eventheader1" : "event1",  
    "eventheader2" : "event2"  
  },  
  "body" : "This is the body1"  
}
```

Till now we haven't actually done anything useful with event headers

## Example 4: event bucketing

```
{  
  "headers" : {  
    "eventheader1" : "event1",  
    "eventheader2" : "event2"  
  },  
  "body" : "This is the body1"  
}
```

The event headers here are just placeholders, they are not being used for any purpose

## Example 4: event bucketing

```
{  
  "headers" : {  
    "eventheader1" : "event1",  
    "eventheader2" : "event2"  
  },  
  "body" : "This is the body1"  
}
```

When you use a HDFS sink, you can use event headers to **dynamically choose the path** where events are written

# Example 4: event bucketing

```
{  
  "headers" : {  
    "eventheader1" : "event1",  
    "eventheader2" : "event2"  
  },  
  "body" : "This is the body1"  
}
```

In http-to-hdfs.properties

We specified a static path where  
the events should be written

```
httpagent.sinks.hdfs-sink.hdfs.path = /tmp/flume/http
```

# Example 4: event bucketing

```
{  
    "headers" : {  
        "eventheader1" : "event1",  
        "eventheader2" : "event2"  
    },  
  
    "body" : "This is the body1"  
}
```

httpagent.sinks.hdfs-sink.hdfs.path = /tmp/flume/http/{topic}

We can tell the sink to choose a different path based on the value of a specific event header

# Example 4: event bucketing

```
{  
    "headers" : {  
        "eventheader1" : "event1",  
        "eventheader2" : "event2"  
    },  
  
    "body" : "This is the body1"  
}
```

httpagent.sinks.hdfs-sink.hdfs.path = /tmp/flume/http/{topic}

The sink will now look for  
an event header called **topic**

# Example 4: event bucketing

```
{  
  "headers" : {  
    "eventheader1" : "event1",  
    "eventheader2" : "event2"  
  },  
  "body" : "This is the body1"  
}
```

httpagent.sinks.hdfs-sink.hdfs.path = /tmp/flume/http/{topic}

Events are bucketed based  
on their topics and written  
to the relevant path

# Example 4: event bucketing

```
httpagent.sinks.hdfs-sink.hdfs.path = /tmp/flume/http/%{topic}
```

```
{  
  "headers" : {"topic" : "topic1"},  
  "body" : "This is the body1"  
}
```

Written to a file in  
`/tmp/flume/http/topic1`

# Example 4: event bucketing

```
httpagent.sinks.hdfs-sink.hdfs.path = /tmp/flume/http/%{topic}
```

```
{  
  "headers" : {"topic" : "topic2"},  
  "body" : "This is the body2"  
}
```

Written to a file in  
*/tmp/flume/http/topic2*

# Example 4: event bucketing

Event bucketing is just 1 example of things you can do with event headers

## Example 4: event bucketing

Flume provides a component called an interceptor

## Example 4: event bucketing

Interceptors can be used to  
process events in Flume  
based on the event headers

More on interceptors later...

**Example 5:**

**Spool to HBase**

## Example 5: Spool to HBase

Let's say you want to  
**migrate data from a**  
legacy data store to HBase

## Example 5: Spool to HBase

You might not want to create  
a new Java application just  
to do the one-time migration

## Example 5: Spool to HBase

You can write a script to  
dump your data as txt files  
in a Spooling directory

## Example 5: Spool to HBase

Flume will take the txt  
files and put the data in  
HBase

## Example 5: Spool to HBase

```
agent1.sources.source1.type = spooldir  
agent1.sources.source1.spoolDir  = /Users/swethakolalapudi/tmp/spooldir
```

We already know how to  
configure the **Spooling  
Directory Source**

# Example 5: Spool to HBase

```
agent1.sinks.hbaseSink.type = org.apache.flume.sink.hbase.AsyncHBaseSink  
agent1.sinks.hbaseSink.table = test_table  
agent1.sinks.hbaseSink.columnFamily = test  
agent1.sinks.hbaseSink.serializer=org.apache.flume.sink.hbase.SimpleAsyncHbaseEventSerializer  
agent1.sinks.hbaseSink.serializer.payloadColumn = pCol
```

The configuration for the  
**HBase sink**

## Example 5: Spool to HBase

```
agent1.sinks.hbaseSink.type = org.apache.flume.sink.hbase.AsyncHBaseSink
```

**AsynchBaseSink** is the  
class name for using  
HBase as sink

## Example 5: Spool to HBase

```
agent1.sinks.hbaseSink.type = org.apache.flume.sink.hbase.AsyncHBaseSink
```

**AsyncHBaseSink** uses an API which allows writing data on multiple threads

## Example 5: Spool to HBase

```
agent1.sinks.hbaseSink.type = org.apache.flume.sink.hbase.AsyncHBaseSink
```

The regular HBase API  
doesn't allow applications  
to write on multiple threads

## Example 5: Spool to HBase

```
agent1.sinks.hbaseSink.type = org.apache.flume.sink.hbase.AsyncHBaseSink
```

The regular HBase API  
doesn't allow applications  
to write on multiple threads

## Example 5: Spool to HBase

```
agent1.sinks.hbaseSink.type = org.apache.flume.sink.hbase.AsyncHBaseSink
```

Flume with AsyncHBaseSink will work better when you want to do bulk imports to HBase

## Example 5:

```
agent1.sinks.hbaseSink.table = test_table  
agent1.sinks.hbaseSink.columnFamily = test
```

Specify the `table` and  
`columnFamily` to which  
you want to write

## Example 5:

```
agent1.sinks.hbaseSink.table = test_table  
agent1.sinks.hbaseSink.columnFamily = test
```

These should already  
exist in HBase, Flume will  
not create them

## Example 5:

```
agent1.sinks.hbaseSink.serializer=org.apache.flume.sink.hbase.SimpleAsyncHbaseEventSerializer
```

Use a serializer to specify  
how the data should be  
written to HBase

## Example 5:

```
agent1.sinks.hbaseSink.serializer=org.apache.flume.sink.hbase.SimpleAsyncHbaseEventSerializer  
agent1.sinks.hbaseSink.serializer.payloadColumn = pCol
```

The SimpleAsyncHbaseEventSerializer  
will write the entire payload to a  
specified column

## Example 5:

```
agent1.sinks.hbaseSink.serializer=org.apache.flume.sink.hbase.SimpleAsyncHbaseEventSerializer  
agent1.sinks.hbaseSink.serializer.payloadColumn = pCol
```

You can implement a custom serializer, if you want to do **custom processing on the payload and insert into specific columns**

## Example 5: Spool to HBase

```
agent1.sinks.hbaseSink.type = org.apache.flume.sink.hbase.AsyncHBaseSink  
agent1.sinks.hbaseSink.table = test_table  
agent1.sinks.hbaseSink.columnFamily = test  
agent1.sinks.hbaseSink.serializer=org.apache.flume.sink.hbase.SimpleAsyncHbaseEventSerializer  
agent1.sinks.hbaseSink.serializer.payloadColumn = pCol
```

Before running, make sure the HBase config file **hbase-site.xml** is available in your classpath

# Example 5: Spool to HBase

```
agent1.sinks.hbaseSink.type = org.apache.flume.sink.hbase.AsyncHBaseSink  
agent1.sinks.hbaseSink.table = test_table  
agent1.sinks.hbaseSink.columnFamily = test  
agent1.sinks.hbaseSink.serializer=org.apache.flume.sink.hbase.SimpleAsyncHbaseEventSerializer  
agent1.sinks.hbaseSink.serializer.payloadColumn = pCol
```

hbase-site.xml

Flume will use this to connect  
to your HBase instance

# Example 6:

HTTP to HDFS and  
Logger

# Example 6: HTTP to HDFS and Logger

So far, we've only seen Flume agents  
with 1 Source, channel sink

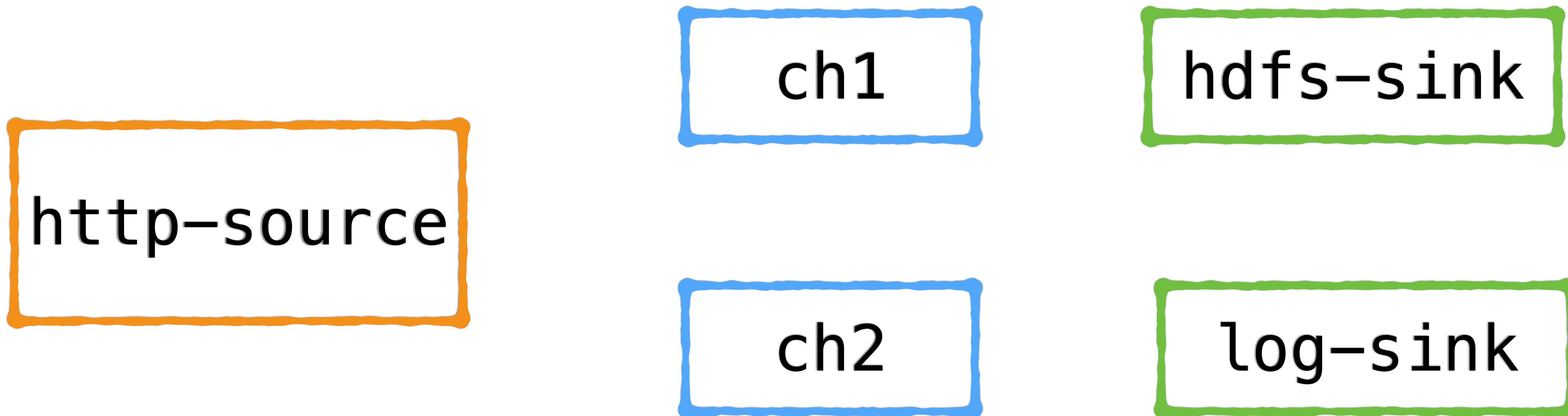
http-source

ch

hdfs-sink

# Example 6: HTTP to HDFS and Logger

A Flume agent that reads from HTTP  
and pushes to both HDFS and console log



```
httpagent.sources = http-source
httpagent.sinks = hdfs-sink log-sink
httpagent.channels = ch1 ch2

httpagent.sources.http-source.channels = ch1 ch2
httpagent.sinks.hdfs-sink.channel = ch1
httpagent.sinks.log-sink.channel = ch2
# Define / Configure Source
#####
httpagent.sources.http-source.type = org.apache.flume.source.http.HTTPSource
httpagent.sources.http-source.bind = localhost
httpagent.sources.http-source.port = 8989
# HDFS File Sink
#####
httpagent.sinks.hdfs-sink.type = hdfs
httpagent.sinks.hdfs-sink.hdfs.path = /tmp/flume/http
httpagent.sinks.hdfs-sink.hdfs.filePrefix = events
httpagent.sinks.hdfs-sink.hdfs.fileSuffix = .log
httpagent.sinks.hdfs-sink.hdfs.inUsePrefix =
httpagent.sinks.hdfs-sink.hdfs.fileType = DataStream
# Logger sink
#####
httpagent.sinks.log-sink.type = logger
# Channels
#####
httpagent.channels.ch1.type = memory
httpagent.channels.ch1.capacity = 1000

httpagent.channels.ch2.type = file
```

http-source

ch1

hdfs-sink

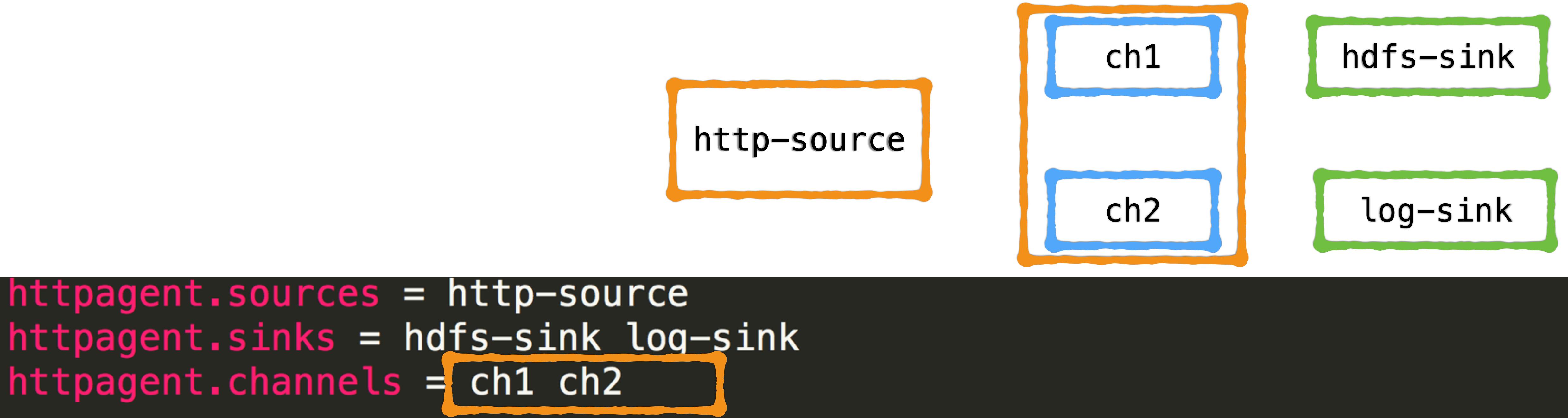
ch2

log-sink

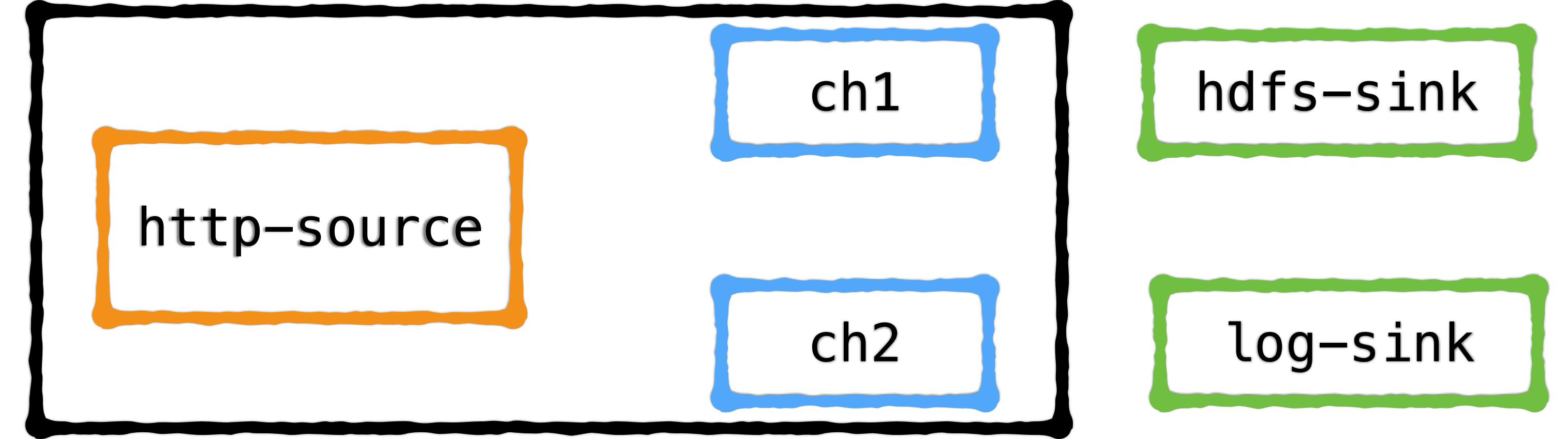


```
httpagent.sources = http-source  
httpagent.sinks = hdfs-sink log-sink  
httpagent.channels = ch1 ch2
```

Set up the sources,  
sinks and channels

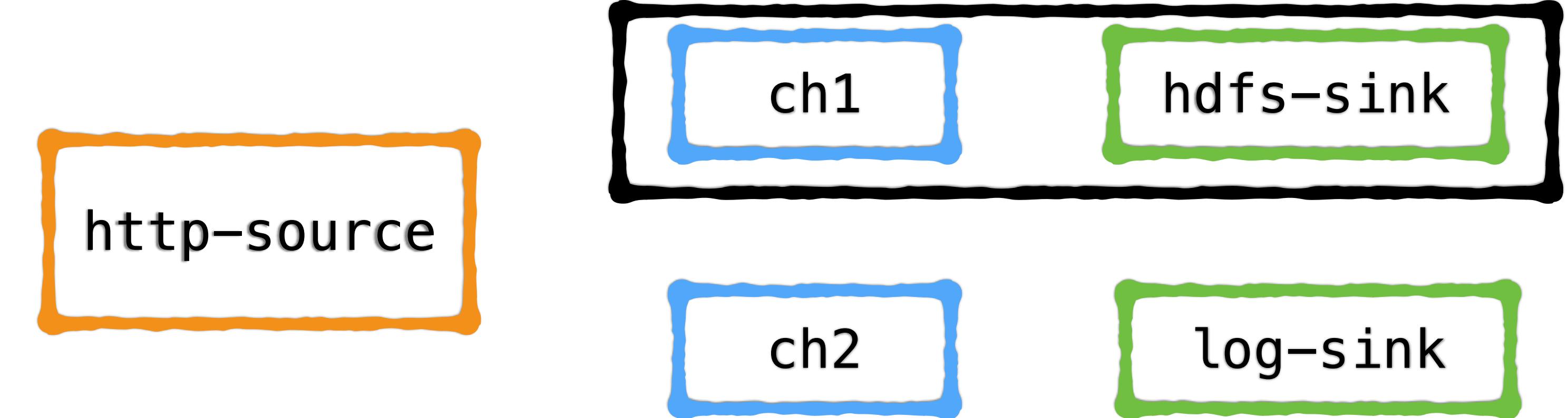


Set up the sources,  
sinks and channels



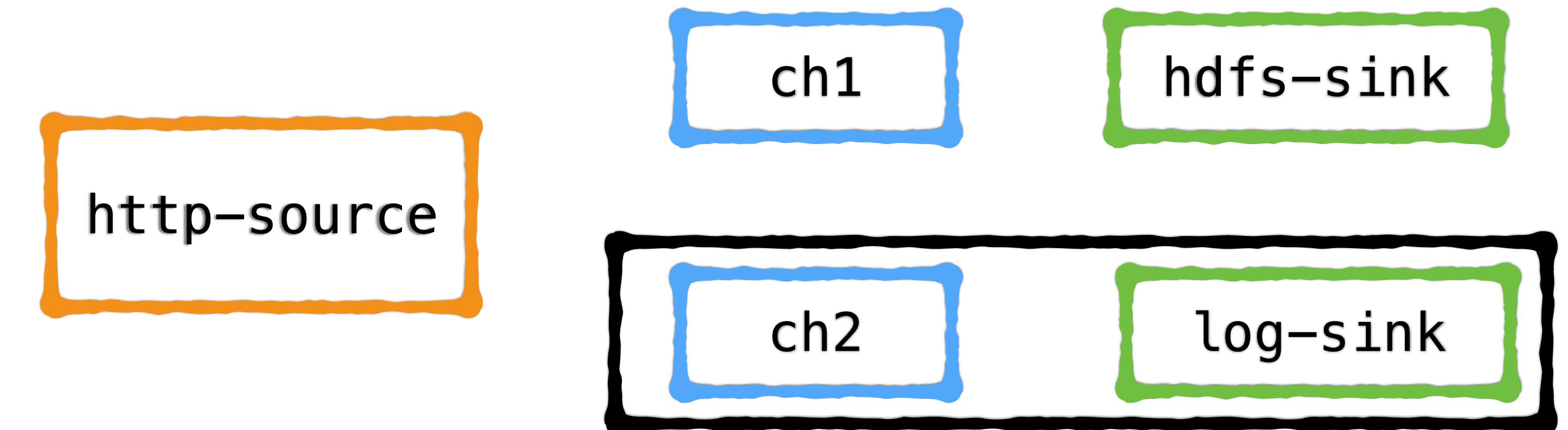
```
httpagent.sources.http-source.channels = ch1 ch2  
httpagent.sinks.hdfs-sink.channel = ch1  
httpagent.sinks.log-sink.channel = ch2
```

Connect the source to  
the channels



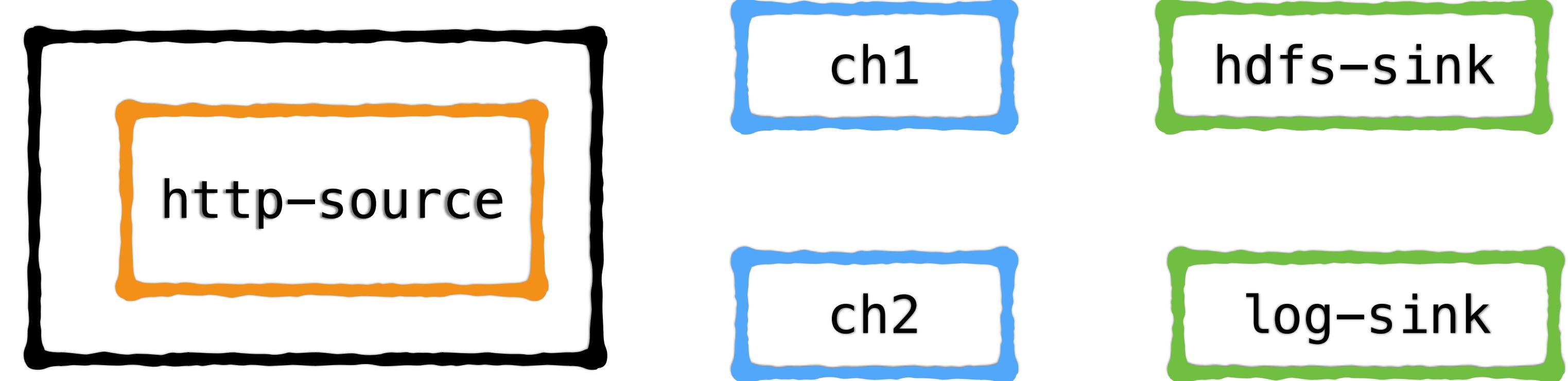
```
httpagent.sources.http-source.channels = ch1 ch2  
httpagent.sinks.hdfs-sink.channel = ch1  
httpagent.sinks.log-sink.channel = ch2
```

# Channel for HDFS sink



```
httpagent.sources.http-source.channels = ch1 ch2
httpagent.sinks.hdfs-sink.channel = ch1
httpagent.sinks.log-sink.channel = ch2
# Define / Configuration
```

# Channel for Logger sink



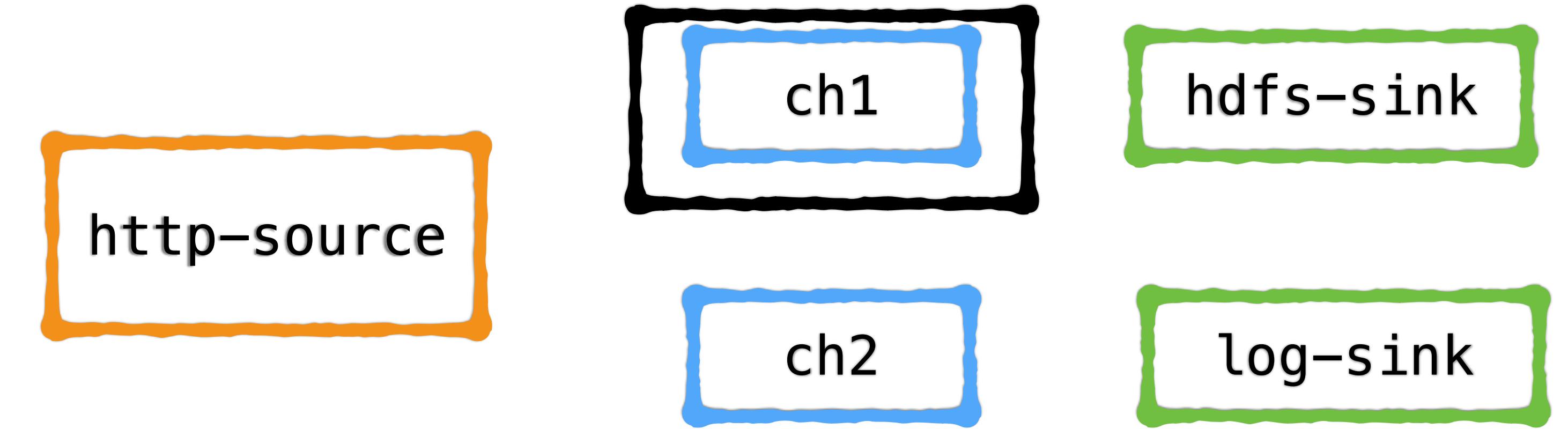
```
# Define / Configure Source
#####
httpagent.sources.http-source.type = org.apache.flume.source.http.HTTPSource
httpagent.sources.http-source.bind = localhost
httpagent.sources.http-source.port = 8989
```

# Configure the HTTP source



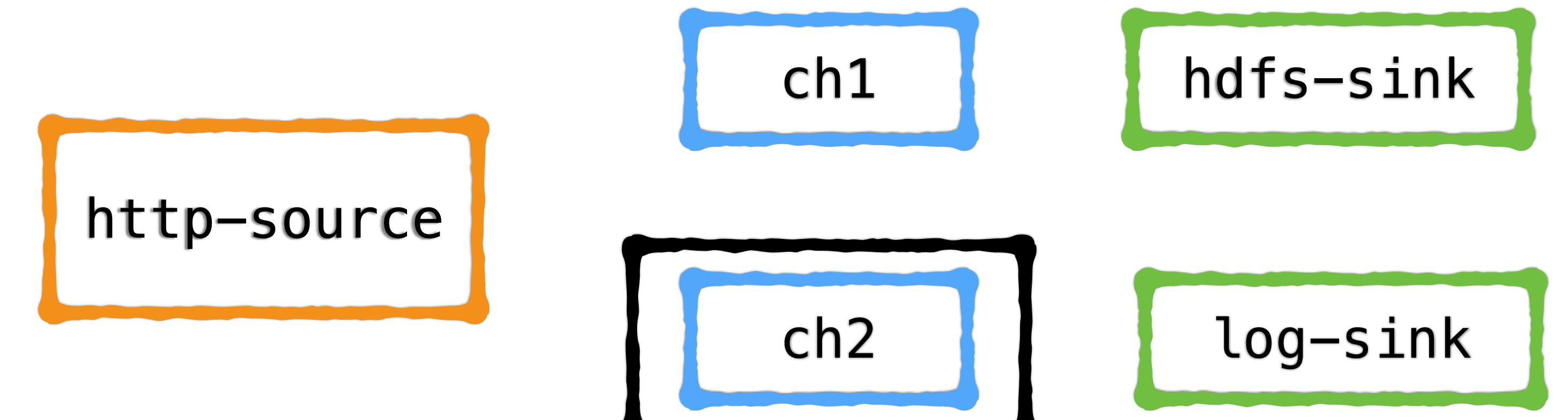
```
# HDFS File Sink
#####
httpagent.sinks.hdfs-sink.type = hdfs
httpagent.sinks.hdfs-sink.hdfs.path = /tmp/flume/http
httpagent.sinks.hdfs-sink.hdfs.filePrefix = events
httpagent.sinks.hdfs-sink.hdfs.fileSuffix = .log
httpagent.sinks.hdfs-sink.hdfs.inUsePrefix =
httpagent.sinks.hdfs-sink.hdfs.fileType = DataStream
```

# Configure the HDFS sink



```
# Channels
#####
httpagent.channels.ch1.type = memory
httpagent.channels.ch1.capacity = 1000
```

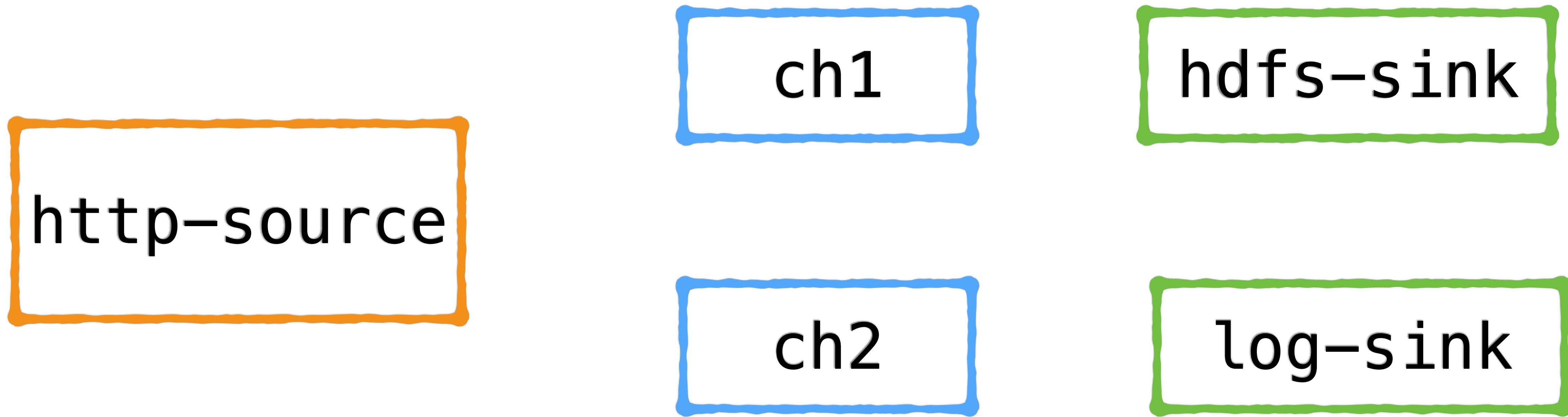
Channel 1 configured as  
memory



# Channel 2 configured as file

```
httpagent.channels.ch2.type = file
```

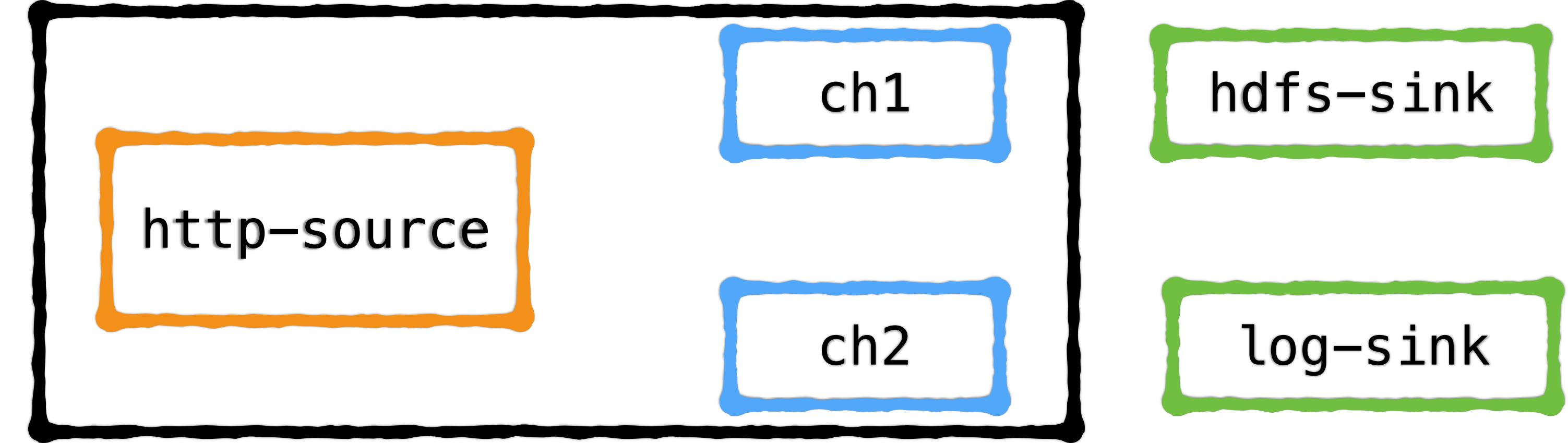
# Example 6: HTTP to HDFS and Logger



When you run, you'll see the output both  
in the console log and the HDFS path

Why?

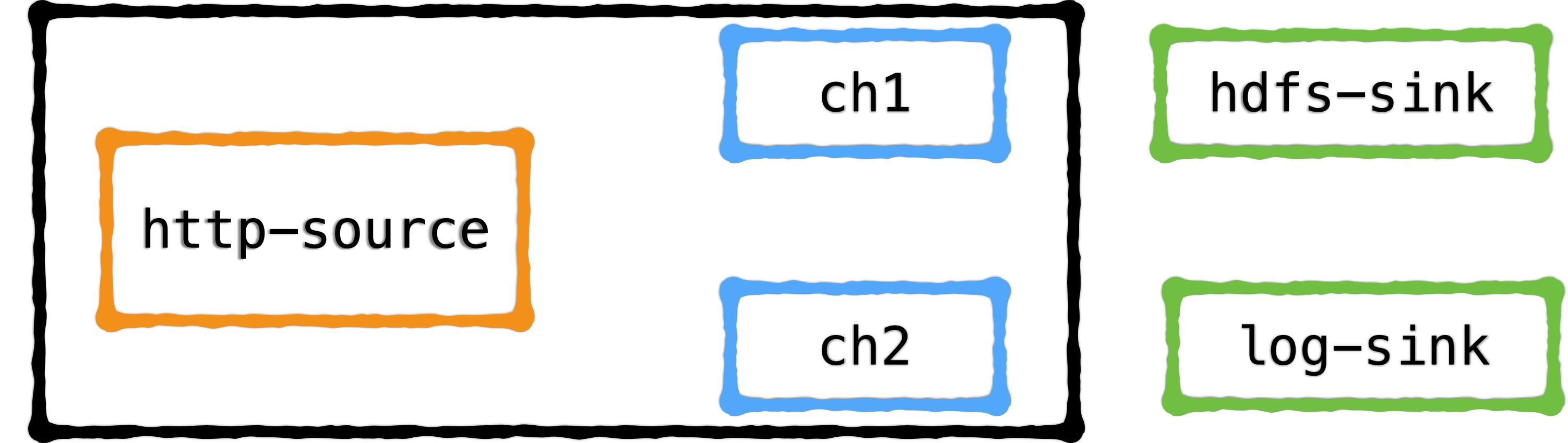
# Why?



```
httpagent.sources.http-source.channels = ch1 ch2  
httpagent.sinks.hdfs-sink.channel = ch1  
httpagent.sinks.log-sink.channel = ch2
```

In this example, the source is connected to 2 channels

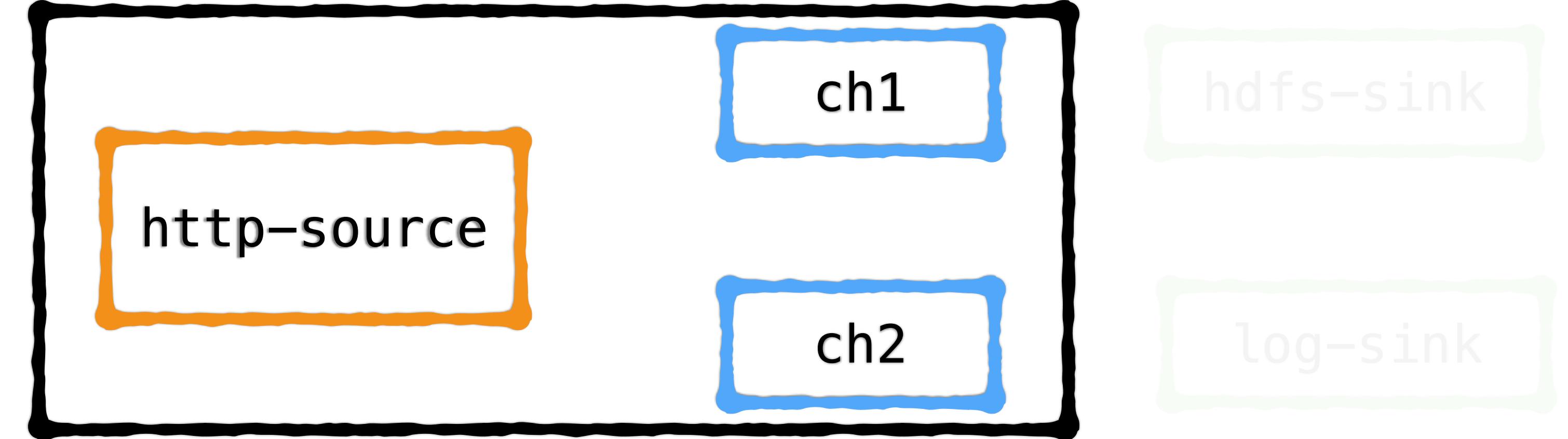
# Why?



```
httpagent.sources.http-source.channels = ch1 ch2  
httpagent.sinks.hdfs-sink.channel = ch1  
httpagent.sinks.log-sink.channel = ch2
```

The decision of which event should be written to which channel is made by a component called **a channel selector**

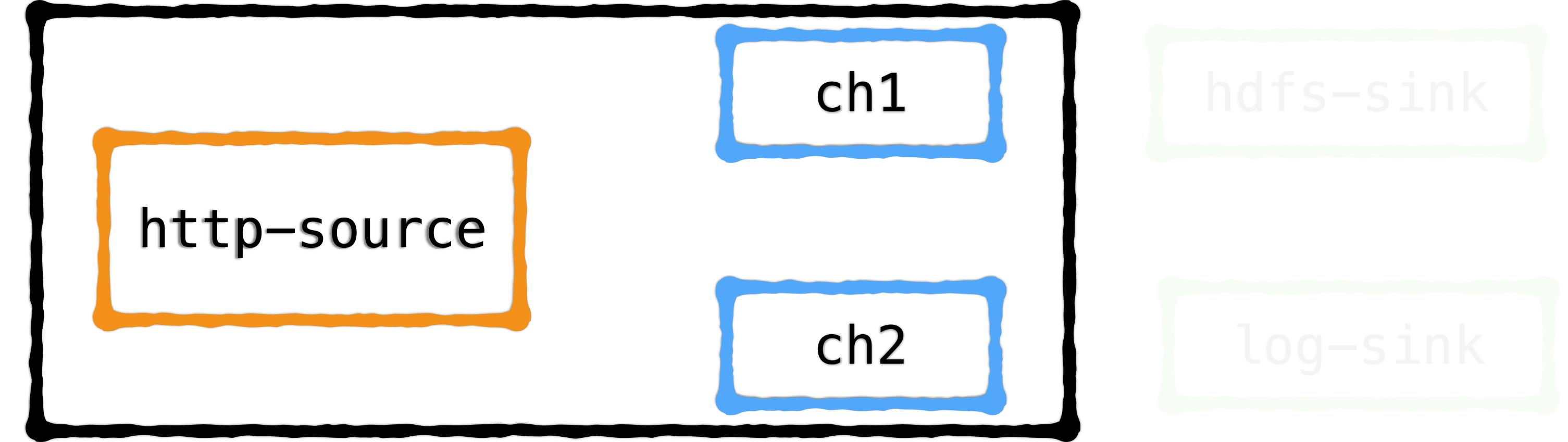
# Why?



## channel selector

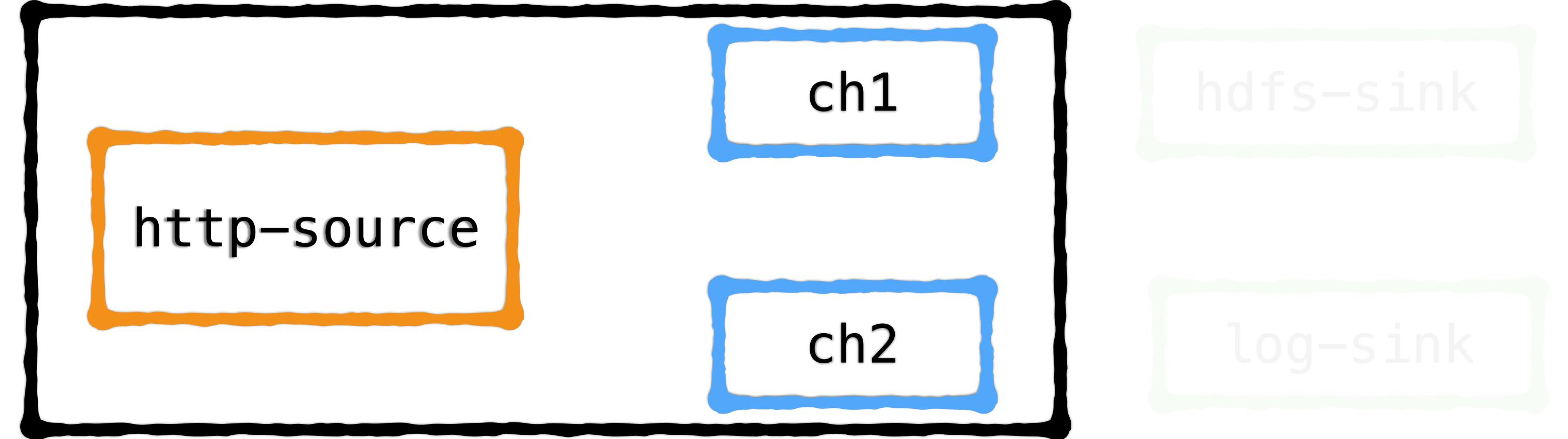
By default all the events from the source are written to both channels

# Why?



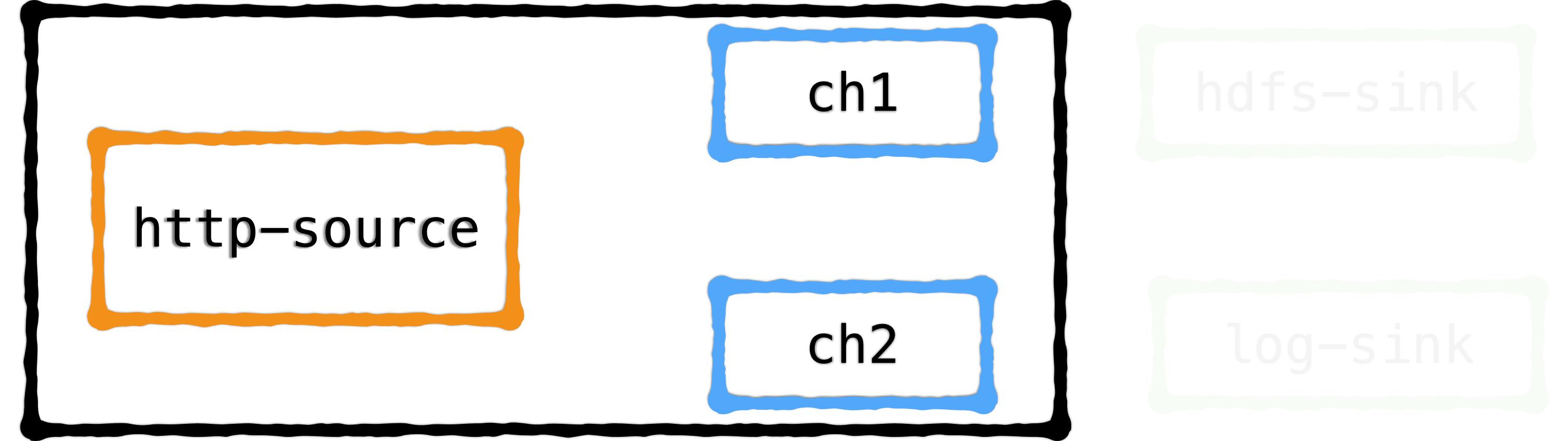
## channel selector

This is called a replicating  
channel selector



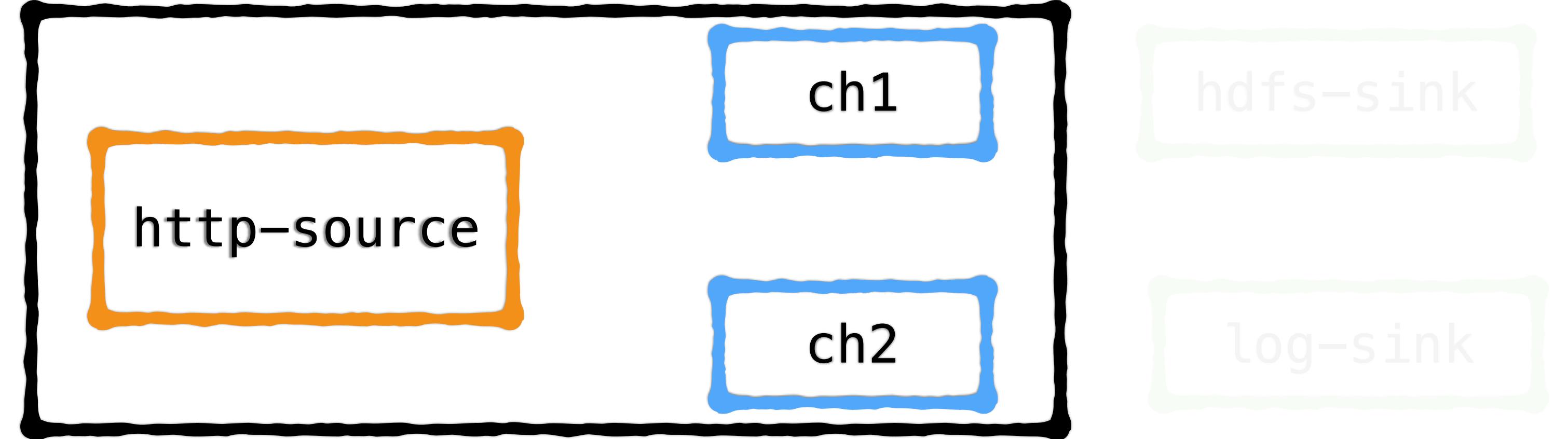
# channel selector

Instead you can use a multiplexing channel selector which routes events to channels based on their headers



# channel selector

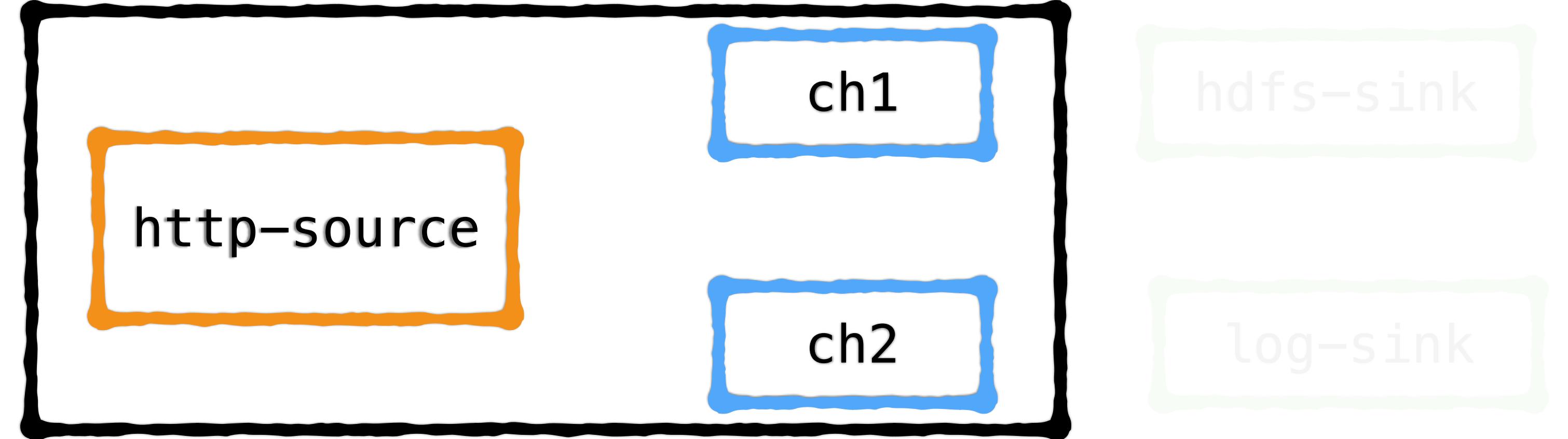
```
# Selector
#####
httpagent.sources.http-source.selector.type = multiplexing
httpagent.sources.http-source.selector.header = show
httpagent.sources.http-source.selector.mapping.1 = ch1
httpagent.sources.http-source.selector.mapping.2 = ch2
```



# channel selector

```
httpagent.sources.http-source.selector.type = multiplexing
```

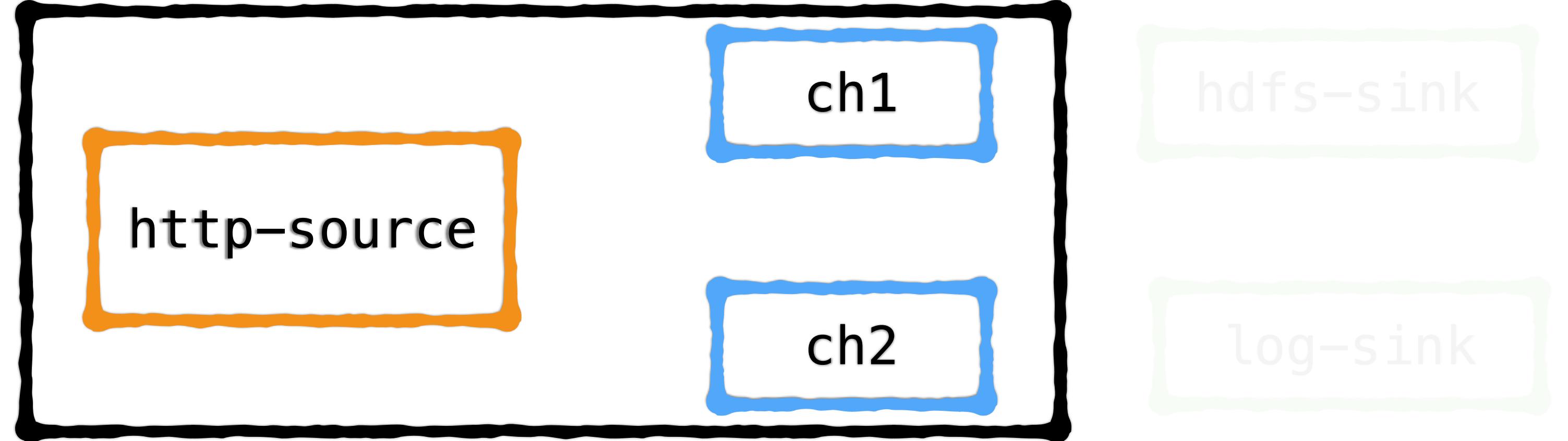
Set the selector type to  
multiplexing



# channel selector

```
httpagent.sources.http-source.selector.header = show
```

Route the events based on  
the value of header “show”



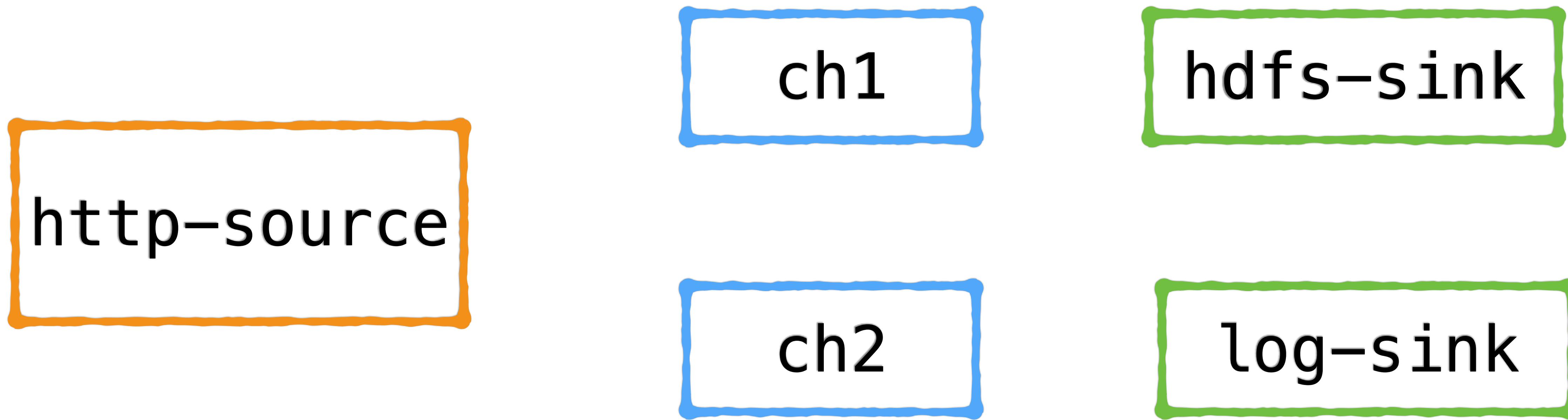
# channel selector

```
httpagent.sources.http-source.selector.mapping.1 = ch1  
httpagent.sources.http-source.selector.mapping.2 = ch2
```

If show = 1, route to ch1

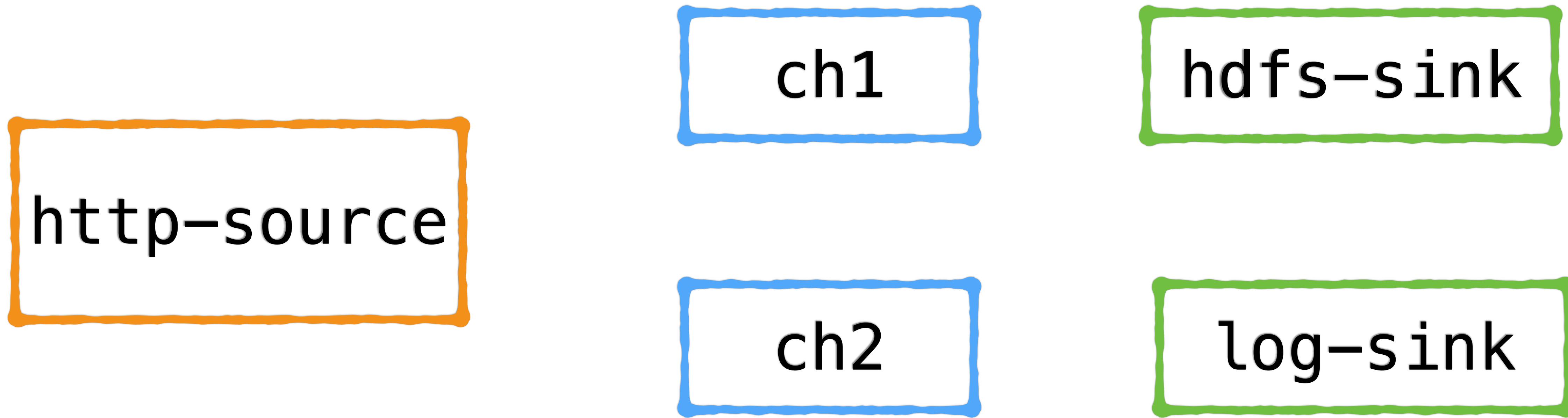
If show = 2, route ch2

## Example 6: HTTP to HDFS and Logger



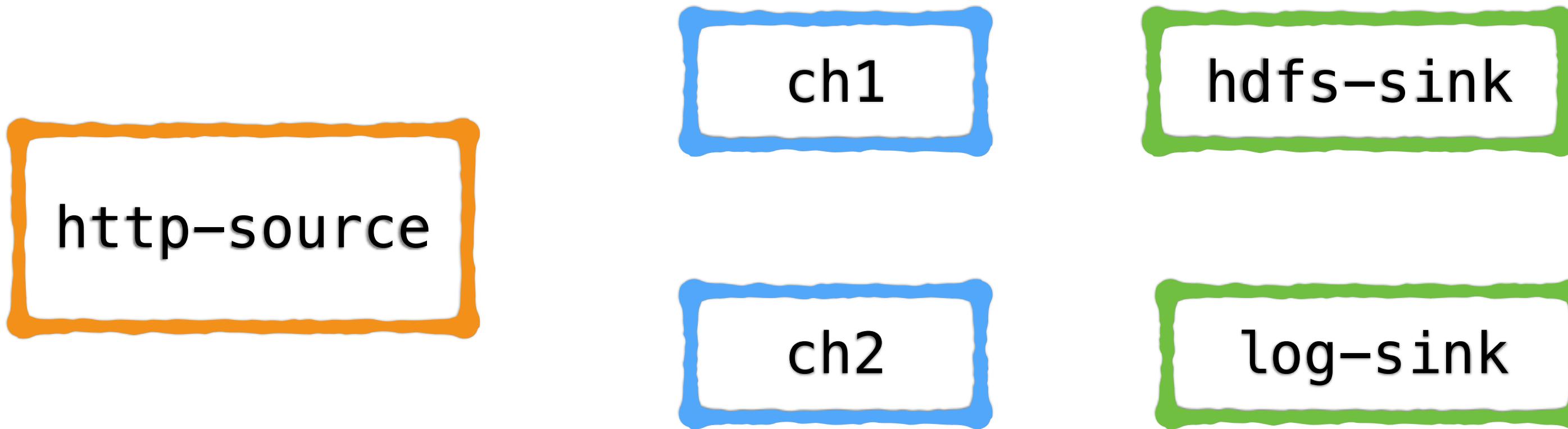
This time when you run, each event will only show up in one of the sinks

## Example 6: HTTP to HDFS and Logger



This time when you run, each event will only show up in one of the sinks

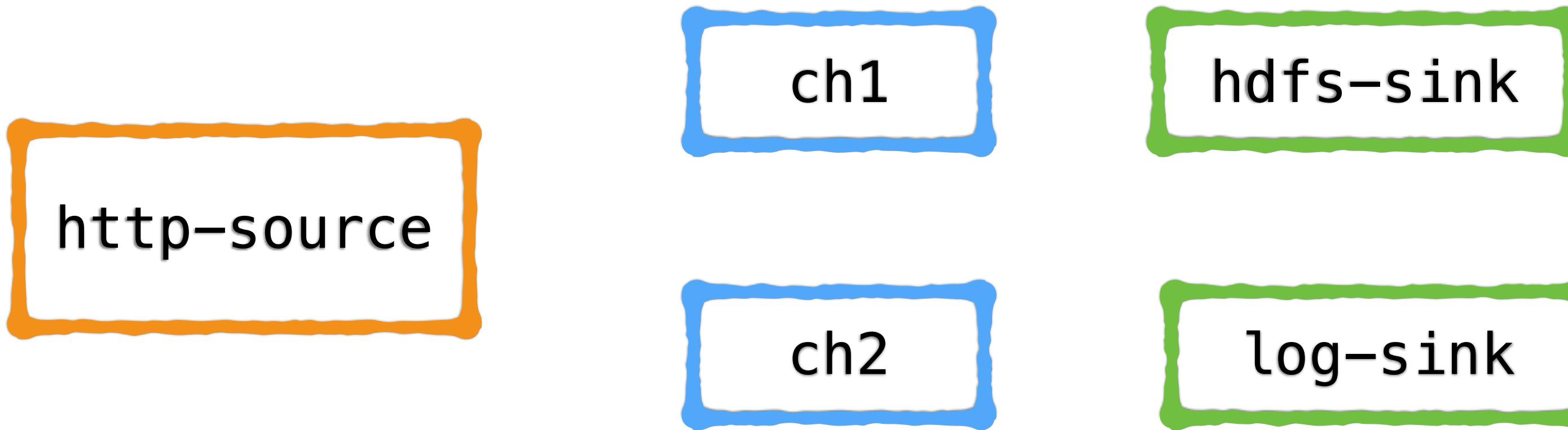
# Example 6: HTTP to HDFS and Logger



```
{  
  "headers" : {"show" : "1"},  
  "body" : "This is the body1"  
}
```

Routed through ch1  
to HDFS

# Example 6: HTTP to HDFS and Logger



```
{  
  "headers" : {"show" : "2"},  
  "body" : "This is the body1"  
}
```

Routed through  
ch2 to logger

Example 7:

Twitter to HDFS

## Example 7: Twitter to HDFS

Say you wanted to monitor  
Twitter for some keywords

And store the tweets in HDFS for  
later analysis

## Example 7: Twitter to HDFS

Flume has a builtin  
Twitter Source

## Example 7: Twitter to HDFS

All you need to do is set up  
a Flume agent with a  
**Twitter Source and specify  
your keywords!**

## Example 7: Twitter to HDFS

```
agent1.sources.source1.type = org.apache.flume.source.twitter.TwitterSource  
agent1.sources.source1.consumerKey = **  
agent1.sources.source1.consumerSecret = **  
agent1.sources.source1.accessToken = **  
agent1.sources.source1.accessTokenSecret = **  
agent1.sources.source1.keywords = @realDonaldTrump, @HillaryClinton
```

Configure the Twitter  
Source

# Example 7: Twitter to HDFS

```
agent1.sources.source1.type = org.apache.flume.source.twitter.TwitterSource
```

The source type is  
**TwitterSource**

## Example 7: Twitter to HDFS

```
agent1.sources.source1.consumerKey = **  
agent1.sources.source1.consumerSecret = **  
agent1.sources.source1.accessToken = **  
agent1.sources.source1.accessTokenSecret = **
```

Specify credentials (these are given when your application interacts with the Twitter API)

# Example 7: Twitter to HDFS

Details    Settings    Keys and Access Tokens    Permissions

## Application Settings

*Keep the "Consumer Secret" a secret. This key should never be human-readable in your application.*

Consumer Key (API Key) [REDACTED]

Consumer Secret (API Secret) [REDACTED]

Access Level    Read and write ([modify app permissions](#))

Owner    hanuraj\_ks

Owner ID    124163864

## Application Actions

[Regenerate Consumer Key and Secret](#)

[Change App Permissions](#)

## Your Access Token

*This access token can be used to make API requests on your own account's behalf. Do not share your access token secret with anyone.*

Access Token [REDACTED]

Access Token Secret [REDACTED]

## Example 7: Twitter to HDFS

```
agent1.sources.source1.consumerKey = **  
agent1.sources.source1.consumerSecret = **  
agent1.sources.source1.accessToken = **  
agent1.sources.source1.accessTokenSecret = **
```

Now Flume can connect  
to the Twitter API

## Example 7: Twitter to HDFS

```
agent1.sources.source1.keywords = @realDonaldTrump, @HillaryClinton
```

Twitter Source accepts a  
comma separated list of  
keywords to search for

## Example 7: Twitter to HDFS

```
agent1.sources.source1.keywords = @realDonaldTrump, @HillaryClinton
```

Any tweets with these keywords will be downloaded and pushed to the sink

## Example 7: Twitter to HDFS

```
agent1.sources.source1.keywords = @realDonaldTrump, @HillaryClinton
```

Let's do 2 things with these tweets

1. **Filter** only those tweets having the word **election**
2. Bucket the events into HDFS directories  
based on the **timestamp** they are downloaded

## Example 7: Twitter to HDFS

1. Filter only those tweets having the word **election**

Normally, systems like Flume  
are only intended for  
transport

## Example 7: Twitter to HDFS

1. Filter only those tweets having the word **election**

Data processing would occur  
at the **source or sink end**

You wouldn't **expect Flume**  
to do any of it

## Example 7: Twitter to HDFS

1. Filter only those tweets having the word **election**

Flume has a component  
called an **interceptor**

## Example 7: Twitter to HDFS

1. Filter only those tweets having the word **election**

Interceptors can process  
events based on their **contents**  
**or headers**

## Example 7: Twitter to HDFS

1. Filter only those tweets having the word **election**

Interceptors give users the  
ability to do **some data**  
**processing within Flume itself**

## Example 7: Twitter to HDFS

1. Filter only those tweets having the word **election**

This is an **overhead** on  
Flume though

## Example 7: Twitter to HDFS

1. Filter only those tweets having the word **election**

Interceptors should be used  
with care to avoid piling up of  
**events in the Flume channel**

## Example 7: Twitter to HDFS

1. Filter only those tweets having the word **election**

One commonly used  
interceptor is a **Regex Filter**  
**Interceptor**

## Example 7: Twitter to HDFS

1. Filter only those tweets having the word election

### Regex Filter Interceptor

This can be used to either include or exclude those events which match a given regex

## Example 7: Twitter to HDFS

1. Filter only those tweets having the word **election**

```
agent1.sources.source1.interceptors = regexInterceptor
```

Interceptors are a Source  
property

## Example 7: Twitter to HDFS

1. Filter only those tweets having the word **election**

```
agent1.sources.source1.interceptors.regexInterceptor.type = regex_filter
```

**regex\_filter** is the alias  
used to set the type

## Example 7: Twitter to HDFS

1. Filter only those tweets having the word **election**

```
agent1.sources.source1.interceptors.regexInterceptor.regex = .*election.*
```

A regular expression to only  
select tweets with the word  
election

## Example 7: Twitter to HDFS

1. Filter only those tweets having the word **election**

```
agent1.sources.source1.interceptors.regexInterceptor.excludeEvents = false
```

**excludeEvents = false**  
only includes those tweets  
which match the regex

## Example 7: Twitter to HDFS

1. Filter only those tweets having the word **election**

```
agent1.sources.source1.interceptors.regexInterceptor.excludeEvents = false
```

**excludeEvents = true**  
would **includes** those tweets  
**which do not match** the regex

## Example 7: Twitter to HDFS

```
agent1.sources.source1.keywords = @realDonaldTrump, @HillaryClinton
```

Let's do 2 things with these tweets

1. **Filter** only those tweets having the word **election**
2. Bucket the events into HDFS directories  
based on the **timestamp** they are downloaded at

## Example 7: Twitter to HDFS

2. Bucket based on the timestamp

HDFS sinks have a property  
called `useLocalTimeStamp`

```
agent1.sinks.sink1.hdfs.useLocalTimeStamp = true
```

Setting to true adds a timestamp  
header to all events in the sink

## Example 7: Twitter to HDFS

### 2. Bucket based on the timestamp

We've already seen how to bucket events based on an event header

```
agent1.sinks.sink1.hdfs.useLocalTimeStamp = true
```

HDFS Sink can parse the individual components of a timestamp header for bucketing

## Example 7: Twitter to HDFS

### 2. Bucket based on the timestamp

```
agent1.sinks.sink1.hdfs.useLocalTimeStamp = true  
agent1.sinks.sink1.hdfs.path = /tmp/flume/twitterinterceptor/%Y/%m/%d/%H
```

HDFS Sink will parse these components from the timestamp header and write the events to the relevant path

# Sqoop

Let's see how to use  
Sqoop to import data  
to HDFS/Hive

# Example 8:

## Sqoop Import from MySQL to HDFS

# Example 8: Sqoop Import from MySQL to HDFS

Let's say you wanted to port  
data from MySQL to HDFS

## Example 8: Sqoop Import from MySQL to HDFS

This could be because

You are porting data from a legacy  
MySQL system to HDFS

- (Or) You use MySQL for transaction processing  
and periodically archive data to HDFS
- (Or) You have a 1 time use case to run  
MapReduce tasks on data in MySQL

## Example 8: Sqoop Import from MySQL to HDFS

In either case, Sqoop is a command line tool that makes it very **easy to bulk-import data from MySQL to HDFS**

## Example 8: Sqoop Import from MySQL to HDFS

In either case, Sqoop is a command line tool that makes it very **easy to bulk-import data from MySQL to HDFS**

## Example 8: Sqoop Import from MySQL to HDFS

Sqoop comes with connectors  
for several kinds of popular  
**RDBMS**

## Example 8: Sqoop Import from MySQL to HDFS

```
sqoop import \
--connect jdbc:mysql://localhost:3306/nseProd \
--username=qt \
--password=password \
--table=tradingDays \
--target-dir /mysql/nseProd \
--m 1
```

# Example 8: Sqoop Import from MySQL to HDFS

```
sqoop import \
```

```
--connect jdbc:mysql://localhost:3306/nseProd \
--username=qt \
--password=password \
--table=tradingDays \
--target-dir /user/qt/nseProd \
--m 1
```

The command to import a  
table or results of a query

# Example 8: Sqoop Import from MySQL to HDFS

```
sqoop import \  
--connect jdbc:mysql://localhost:3306/nseProd \  
--username=qt \  
--password=password \  
--table=tradingDays \  
--target-dir /mysql/nseProd \  
--m 1
```

It launches a MapReduce task  
on the Hadoop cluster to import  
and write the data to HDFS

# Example 8: Sqoop Import from MySQL to HDFS

```
sqoop import \
--connect jdbc:mysql://localhost:3306/nseProd \
--username=qt \
--password=password \
--table=tradingDays \
--target-dir /mysql/nseProd \
--m 1
```

How to connect to  
MySQL

# Example 8: Sqoop Import from MySQL to HDFS

```
sqoop import \
--connect jdbc:mysql://localhost:3306/nseProd \
--username=qt \
--password=password \
--table=tradingDays \
--target-dir /mysql/nseProd \
--m 1
```

How to connect to the  
RDBMS

# Example 8: Sqoop Import from MySQL to HDFS

```
sqoop import \
--connect jdbc:mysql://localhost:3306/nseProd \
--username=qt \
--password=password \
--table=tradingDays \
--target-dir /mysql/nseProd \
--m 1
```

The connector and  
database type

# Example 8: Sqoop Import from MySQL to HDFS

```
sqoop import \  
--connect jdbc:mysql://localhost:3306/nseProd \  
--username=qt \  
--password=password \  
--table=tradingDays \  
--target-dir /mysql/nseProd \  
--m 1
```

Host and port of the  
MySQL server

# Example 8: Sqoop Import from MySQL to HDFS

```
sqoop import \
--connect jdbc:mysql://localhost:3306/nseProd \
--username=qt \
--password=password \
--table=tradingDays \
--target-dir /mysql/nseProd \
--m 1
```

The name of the  
database

# Example 8: Sqoop Import from MySQL to HDFS

```
sqoop import \
--connect jdbc:mysql://localhost:3306/nseProd \
--username=qt \
--password=password \
--table=tradingDays \
--target-dir /mysql/nseProd \
--m 1
```

Credentials to connect  
to the database

# Example 8: Sqoop Import from MySQL to HDFS

```
sqoop import \
--connect jdbc:mysql://localhost:3306/nseProd \
--username=qt \
--password=password \
--table=tradingDays \
--target-dir /mysql/nseProd \
--m 1
```

The table to be imported

# Example 8: Sqoop Import from MySQL to HDFS

```
--query 'select year, month, day from  
tradingDays where year=2016 and $CONDITIONS'  
sqoop import \--connect jdbc:mysql://localhost:3306/nseProd \--username=qt \--password=password \--table=tradingDays \--target-dir /mysql/nseProd \--m 1
```

We can specify a query and conditions for the data instead

## Example 8: Sqoop Import from MySQL to HDFS

```
--query 'select year, month, day from  
tradingDays where year=2016 and $CONDITIONS'
```

You can specify any query, but end  
it with where \$CONDITIONS

# Example 8: Sqoop Import from MySQL to HDFS

```
sqoop import \  
--connect jdbc:mysql://localhost:3306/nseProd \  
--username=qt \  
--password=password \  
--table=tradingDays \  
--target-dir /mysql/nseProd \  
--m 1
```

This option is mandatory if the table you are importing doesn't have a primary key, or if you are importing a query

# Example 8: Sqoop Import from MySQL to HDFS

```
sqoop import \n--connect jdbc:mysql://localhost:3306/nseProd \n--username=qt \n--password=password \n--table=tradingDays \n--target-dir /mysql/nseProd \n
```

**--m 1**

Sqoop uses the primary key to decide how many mappers to use, and for splitting the rows among mappers

# Example 8: Sqoop Import from MySQL to HDFS

```
sqoop import \n--connect jdbc:mysql://localhost:3306/nseProd \n--username=qt \n--password=password \n--table=tradingDays \n--target-dir /mysql/nseProd \n
```

When there is no primary key

**--m 1**

We can use this option  
to explicitly set the  
number of mappers to 1

# Example 8: Sqoop Import from MySQL to HDFS

```
sqoop import \  
--connect jdbc:mysql://localhost:3306/nseProd \  
--username=qt \  
--password=password \  
--table=tradingDays \  
--target-dir /mysql/nseProd
```

**--m** **1**

**Alternatively**  
**use the split-by option to**  
**explicitly specify the**  
**column to use for splitting**  
**rows between mappers**

## Example 8: Sqoop Import from MySQL to HDFS

```
sqoop import \  
---connect jdbc:mysql://localhost:3306/nseProd \  
---username=qt \  
---password=password \  
---table=tradingDays \  
--target-dir /mysql/nseProd \  
--m 1
```

The target directory to  
which the table/result  
should be imported

# Example 8: Sqoop Import from MySQL to HDFS

```
sqoop import \
--connect jdbc:mysql://localhost:3306/nseProd \
--username=qt \
--password=password \
--table=tradingDays \
--target-dir /mysql/nseProd \
--m 1
```

The data will be imported to  
a file/files in this directory

# Example 9:

## Sqoop Import from MySQL to Hive

# Example 9: Sqoop Import from MySQL to Hive

What would you do with data  
imported from MySQL to HDFS?

# Example 9: Sqoop Import from MySQL to Hive

What would you do with data imported from MySQL to HDFS?

One option is to run MapReduce tasks to analyze and process the data

## Example 9: Sqoop Import from MySQL to Hive

Hive provides a SQL like interface to data stored in HDFS

You could create a table in Hive and point it to the files imported by Sqoop

# Example 9: Sqoop Import from MySQL to Hive

You could **create a table in Hive and point it to the files imported by Sqoop**

This involves 2 steps

1. Import a table to HDFS
2. Create a Hive table and point it to a HDFS directory

# Example 9: Sqoop Import from MySQL to Hive

Instead, Sqoop can do  
both steps with 1  
command

# Example 9: Sqoop Import from MySQL to Hive

```
sqoop import \
--connect jdbc:mysql://localhost:3306/nseProd \
--username=qt \
--password=password \
--table=tradingDays \
--hive-import \
--hive-table=tradingDays \
--target-dir /mysql/table/tradingDays2 \
--m 1
```

# Example 9: Sqoop Import from MySQL to Hive

```
sqoop import \  
--connect jdbc:mysql://localhost:3306/nseProd \  
--username=qt \  
--password=password \  
--table=tradingDays \  
--hive-import \  
--hive-table=tradingDays \  
--target-dir /mysql/table/tradingDays2 \  
--m 1
```

This is the same as if you were  
importing the table tradingDays to HDFS

# Example 9: Sqoop Import from MySQL to Hive

```
sqoop import \
--connect jdbc:mysql://localhost:3306/nseProd \
--username=qt \
--password=password \
--table=tradingDays \
--hive-import \
--hive-table=tradingDays \
--target-dir /mysql/table/tradingDays2 \
--m 1
```

Specifies that its a hive import

# Example 9: Sqoop Import from MySQL to Hive

```
sqoop import \  
--connect jdbc:mysql://localhost:3306/nseProd \  
--username=qt \  
--password=password \  
--table=tradingDays \  
--hive-import \  
--hive-table=tradingDays \  
--target-dir /mysql/table/tradingDays2 \  
--m 1
```

The name of the table in Hive

# Example 9: Sqoop Import from MySQL to Hive

```
sqoop import \  
--connect jdbc:mysql://localhost:3306/nseProd \  
--username=qt \  
--password=password \  
--table=tradingDays \  
--hive-import \  
--hive-table=tradingDays \  
--target-dir /mysql/table/tradingDays2 \  
--m 1
```

This table doesn't have to exist in  
Hive

# Example 9: Sqoop Import from MySQL to Hive

```
sqoop import \  
--connect jdbc:mysql://localhost:3306/nseProd \  
--username=qt \  
--password=password \  
--table=tradingDays \  
--hive-import \  
--hive-table=tradingDays \  
--target-dir /mysql/table/tradingDays2 \  
--m 1
```

Sqoop will create an external table for you  
and point the table to the directory in HDFS

# Example 10:

## Sqoop Incremental Import

## Example 10: Sqoop Incremental Import

Let's say you have a sales application that uses MySQL for transaction processing

You periodically archive the MySQL data to HDFS/Hive

# Example 10: Sqoop Incremental Import

The idea is that you want

MySQL to serve your transactional needs

HDFS/Hive for analytical processing needs

# Example 10: Sqoop Incremental Import

Periodically, say

every day/every week

Import new data from MySQL to HDFS

## Example 10: Sqoop Incremental Import

Import new data from MySQL to HDFS

Instead of importing the entire  
table every day/week

Import only new data  
added since last import

# Example 10: Sqoop Incremental Import

Import only new data added since last import

This is called an  
incremental import

## Example 10: Sqoop Incremental Import

Sqoop allows you to save  
an import with specific  
options as a Job

## Example 10: Sqoop Incremental Import

Job

Jobs help you perform  
repetitive imports by saving all  
the options under a given name

## Example 10: Sqoop Incremental Import

Job

You can specify all the regular options like connect, table, target-dir etc

## Example 10: Sqoop Incremental Import

Job

Specify additional  
options for **incremental**  
**import**

# Example 10: Sqoop Incremental Import

```
sqoop import \
--connect jdbc:mysql://localhost:3306/nseProd \
--username=qt \
--password=password \
--table=tradingDays \
--target-dir /mysql/nseProd \
--m 1
```

An import for importing a  
table to HDFS

# Example 10: Sqoop Incremental Import

```
sqoop job \
[--create myjob \
--import \
--connect jdbc:mysql://localhost:3306/nseProd \
--username=qt \
--password=password \
--table=tradingDays \
--target-dir /mysql/nseProd \
--m 1
```

Save this import as a job

# Example 10: Sqoop Incremental Import

```
sqoop job \
--create myjob \
--import \
--connect jdbc:mysql://localhost:3306/nseProd \
--username=tiger \
--password=password \
--table=tradingDays \
--target-dir /mysql/nseProd \
--m 1
```

**Make this an incremental import job**

```
--incremental lastmodified
--check-column ts
```

# Example 10: Sqoop Incremental Import

```
sqoop job \
--create myjob \
--import \
--connection=mysql://192.168.1.106/nseProject \
--username=nseuser \
--password=password \
--table=trades \
--target-dir=/user/nseuser/trades \
--m 1
```

This specifies that the job is  
an incremental import

```
--incremental lastmodified
--check-column ts
```

# Example 10: Sqoop Incremental Import

```
sqoop job \
--create myjob \
--import \
--connect jdbc:mysql://localhost:3306/nseProd \
--username nseprod \
--password nseprod \
--table stocks \
--target-dir /mysql/nseProd \
--m 1
```

**--incremental lastmodified**

**-check-column ts**

# Example 10: Sqoop Incremental Import

```
sqoop job \
--create myjob \
--import \
--connect jdbc:mysql://localhost:3306/nseProd \
--username nse \
--password password \
--table nse \
--target-dir /mysql/nseProd \
--m 1
```

This specifies a column in the table which helps identify newly added/modified rows

**--incremental lastmodified**  
**-check-column ts**

# Example 10: Sqoop Incremental Import

```
sqoop job \
--create myjob \
--import \
--connect jdbc:mysql://localhost:3306/nseProd \
--username=nseProd \
--password=pASSWORD \
--table=stocks \
--target-dir /mysql/nseProd \
--m 1
```

The column should ideally be a timestamp  
that's updated whenever the row is changed

```
--incremental lastmodified
--check-column ts
```

# Example 10: Sqoop Incremental Import

Every time the job is run, the max value of the ts column is saved

lastvalue

--incremental lastmodified

-check-column ts

# Example 10: Sqoop Incremental Import

```
sqoop job \
--create myjob \
--import \
--connect jdbc:mysql://localhost:3306/nseProd \
--username=nseprod \
--password=password \
--table=stocks \
--target-dir /mysql/nseProd \
--m 1
```

The next time the job is run, only rows where `ts > lastvalue` are retrieved

```
--incremental lastmodified
--check-column ts
```

# Example 10: Sqoop Incremental Import

```
sqoop job \
--create myjob \
--import \
--connect jdbc:mysql://localhost:3306/nseProd \
--username=qt \
--password=password \
--table=tradingDays \
--target-dir /mysql/nseProd \
--m 1
```

To run the job  
sqoop job --exec myjob