Web Crawling

Paolo Boldi
DSI
LAW (Laboratory for Web Algorithmics)
Università degli Studi di Milan

Looking for information?

Looking for information is becoming more and more difficult, for various reasons:

Looking for information?

Looking for information is becoming more and more difficult, for various reasons:

- dimension (*too much information!* . . .)

Looking for information?

Looking for information is becoming more and more difficult, for various reasons:

- dimension (*too much information!* ...)
- lack of semantic information (attempts to move to the *Semantic Web*) and structure

Looking for information?

Looking for information is becoming more and more difficult, for various reasons:

- dimension (*too much information!* ...)
- lack of semantic information (attempts to move to the *Semantic Web*) and structure
- information quality is extremely heterogeneous

Looking for information?

Looking for information is becoming more and more difficult, for various reasons:

- dimension (*too much information!* ...)
- lack of semantic information (attempts to move to the *Semantic Web*) and structure
- information quality is extremely heterogeneous
- documents rapidly become unavailable, obsolete, modified...

Looking for information?

Looking for information is becoming more and more difficult, for various reasons:

- dimension (*too much information!* ...)
- lack of semantic information (attempts to move to the *Semantic Web*) and structure
- information quality is extremely heterogeneous
- documents rapidly become unavailable, obsolete, modified...

...about 80% of users make use of a search engine to try to look for information

Structure of a search engine

A *search engine* is conceptually made of three components:

Structure of a search engine

A *search engine* is conceptually made of three components:

- a web crawler (or web spider)

Structure of a search engine

A *search engine* is conceptually made of three components:

- a web crawler (or web spider)
- an indexing engine

Structure of a search engine

A *search engine* is conceptually made of three components:

- a web crawler (or web spider)
- an indexing engine
- a front end

Usually, the end user only knows the latter component, that is, the part of a search engine that actually answers users' queries.

Web Crawler

This component aims at visiting the web (or a certain portion of it) and at gathering locally (information about) pages that are found, saving this information in some structure (called a *store*) that allows the other phases to be performed.

Web Crawler

This component aims at visiting the web (or a certain portion of it) and at gathering locally (information about) pages that are found, saving this information in some structure (called a *store*) that allows the other phases to be performed.

This is the component that we are going to describe in the rest of this talk. Writing a web crawler is apparently an easy task (from an algorithmic viewpoint), but some issues should be considered carefully (and make the problem less trivial):

Web Crawler

This component aims at visiting the web (or a certain portion of it) and at gathering locally (information about) pages that are found, saving this information in some structure (called a *store*) that allows the other phases to be performed.

This is the component that we are going to describe in the rest of this talk. Writing a web crawler is apparently an easy task (from an algorithmic viewpoint), but some issues should be considered carefully (and make the problem less trivial):

- Bandwidth

Web Crawler

This component aims at visiting the web (or a certain portion of it) and at gathering locally (information about) pages that are found, saving this information in some structure (called a *store*) that allows the other phases to be performed.

This is the component that we are going to describe in the rest of this talk. Writing a web crawler is apparently an easy task (from an algorithmic viewpoint), but some issues should be considered carefully (and make the problem less trivial):

- Bandwidth
- Refresh policies

Web Crawler

This component aims at visiting the web (or a certain portion of it) and at gathering locally (information about) pages that are found, saving this information in some structure (called a *store*) that allows the other phases to be performed.

This is the component that we are going to describe in the rest of this talk. Writing a web crawler is apparently an easy task (from an algorithmic viewpoint), but some issues should be considered carefully (and make the problem less trivial):

- Bandwidth
- Refresh policies
- Presence of hidden material (the "dark Web")

Web Crawler

This component aims at visiting the web (or a certain portion of it) and at gathering locally (information about) pages that are found, saving this information in some structure (called a *store*) that allows the other phases to be performed.

This is the component that we are going to describe in the rest of this talk. Writing a web crawler is apparently an easy task (from an algorithmic viewpoint), but some issues should be considered carefully (and make the problem less trivial):

- Bandwidth
- Refresh policies
- Presence of hidden material (the “dark Web”)
- Failure to respect standards (HTTP, HTML, etc.)

Web Crawler

This component aims at visiting the web (or a certain portion of it) and at gathering locally (information about) pages that are found, saving this information in some structure (called a *store*) that allows the other phases to be performed.

This is the component that we are going to describe in the rest of this talk. Writing a web crawler is apparently an easy task (from an algorithmic viewpoint), but some issues should be considered carefully (and make the problem less trivial):

- Bandwidth
- Refresh policies
- Presence of hidden material (the "dark Web")
- Failure to respect standards (HTTP, HTML, etc.)
- Spider traps!

Indexing engine

This is a set of tools that analyze the store (where the visited pages have been stored) and produces a set of addictional structures (generically called *indices*) that allow one to access rapidly the store, and to answer efficiently to users' queries.

Indexing engine

This is a set of tools that analyze the store (where the visited pages have been stored) and produces a set of addictional structures (generically called *indices*) that allow one to access rapidly the store, and to answer efficiently to users' queries.

Among the purposes of this phase:

Indexing engine

This is a set of tools that analyze the store (where the visited pages have been stored) and produces a set of addictional structures (generically called *indices*) that allow one to access rapidly the store, and to answer efficiently to users' queries.

Among the purposes of this phase:

- Extract textual information from the pages (parsing)

Indexing engine

This is a set of tools that analyze the store (where the visited pages have been stored) and produces a set of addictional structures (generically called *indices*) that allow one to access rapidly the store, and to answer efficiently to users' queries.

Among the purposes of this phase:

- Extract textual information from the pages (parsing)
- Detecting the presence of duplicates (or quasi-duplicates), due to mirroring or other phenomena

Indexing engine

This is a set of tools that analyze the store (where the visited pages have been stored) and produces a set of addictional structures (generically called *indices*) that allow one to access rapidly the store, and to answer efficiently to users' queries.

Among the purposes of this phase:

- Extract textual information from the pages (parsing)
- Detecting the presence of duplicates (or quasi-duplicates), due to mirroring or other phenomena
- Detecting the presence of spamming

Indexing engine

This is a set of tools that analyze the store (where the visited pages have been stored) and produces a set of addictional structures (generically called *indices*) that allow one to access rapidly the store, and to answer efficiently to users' queries.

Among the purposes of this phase:

- Extract textual information from the pages (parsing)
- Detecting the presence of duplicates (or quasi-duplicates), due to mirroring or other phenomena
- Detecting the presence of spamming
- Produce suitable (inverted) indices, and compute information needed in the last phase for selection and ranking purposes

Front end

Answers the users' queries. Given a query, it has to *select* the set of pages that match the query and it has to *rank* them (i.e., to decide in which order the selected pages should be presented).

Front end

Answers the users' queries. Given a query, it has to *select* the set of pages that match the query and it has to *rank* them (i.e., to decide in which order the selected pages should be presented). Both tasks are carried out using the indices produced in the previous phase.

Front end (cont'd)

Typically, a search engine also provides additional features, like:

Front end (cont'd)

Typically, a search engine also provides additional features, like:

- Sophisticated searches (boolean operators, proximity operators, site- or language-restricted searches etc.)

Front end (cont'd)

Typically, a search engine also provides additional features, like:

- Sophisticated searches (boolean operators, proximity operators, site- or language-restricted searches etc.)
- Usage of ontologic suggestions ("windows" is an operating system or an opening in a wall?)

Front end (cont'd)

Typically, a search engine also provides additional features, like:

- Sophisticated searches (boolean operators, proximity operators, site- or language-restricted searches etc.)
- Usage of ontologic suggestions ("windows" is an operating system or an opening in a wall?)
- Linguistic recognition, stemming and hyper/hyponym substitution ("flat" expands to "flat OR flats OR apartments OR houses" etc.)

Front end (cont'd)

Typically, a search engine also provides additional features, like:

- Sophisticated searches (boolean operators, proximity operators, site- or language-restricted searches etc.)
- Usage of ontologic suggestions ("windows" is an operating system or an opening in a wall?)
- Linguistic recognition, stemming and hyper/hyponym substitution ("flat" expands to "flat OR flats OR apartments OR houses" etc.)
- User profiling (through query/bookmark logging etc.)

Front end (cont'd)

Typically, a search engine also provides additional features, like:

- Sophisticated searches (boolean operators, proximity operators, site- or language-restricted searches etc.)
- Usage of ontologic suggestions ("windows" is an operating system or an opening in a wall?)
- Linguistic recognition, stemming and hyper/hyponym substitution ("flat" expands to "flat OR flats OR apartments OR houses" etc.)
- User profiling (through query/bookmark logging etc.)
- Automatic clustering and classification

Structure of a crawler (I)

A crawler (aka spider) must harvest pages, essentially by performing a visit of the Web graph.

Structure of a crawler (I)

A crawler (aka spider) must harvest pages, essentially by performing a visit of the Web graph.

The crawl is performed starting from one (or more) page(s), called *crawl seed*.

Structure of a crawler (II)

A visit:

Structure of a crawler (II)

A visit:

$\mathrm{visit}()$: **Perform a visit cycle**
while $F \neq \emptyset$ **do**
  $x \leftarrow F.\mathrm{pick}()$
  $\mathrm{visit}(x)$ i.e.: fetch the page with URL x!
  $St[x] \leftarrow \mathrm{black}$
  for $y \in N^+(x)$ **do**
    if $S[y] = \mathrm{white}$ **then**
      $St[y] \leftarrow \mathrm{grey}$
      $F.\mathrm{add}(y)$
    end if
  end for
end while

Problems

Choosing the seed

Problems

Choosing the seed

- Determines the set of pages that will be visited (*coverage*)

Problems

Choosing the seed

- Determines the set of pages that will be visited (*coverage*)
- If you start from any page in the giant component, you will (theoretically) end up visiting the whole web, except essentially from the source components

Problems

Choosing the seed

- Determines the set of pages that will be visited (*coverage*)
- If you start from any page in the giant component, you will (theoretically) end up visiting the whole web, except essentially from the source components
- There is a (ever growing) portion of the web that is not reachable simply through links: this is the so called "dark matter" (automatic form-filling? fake user registration?)

Problems

Choosing the seed

- Determines the set of pages that will be visited (*coverage*)
- If you start from any page in the giant component, you will (theoretically) end up visiting the whole web, except essentially from the source components
- There is a (ever growing) portion of the web that is not reachable simply through links: this is the so called "dark matter" (automatic form-filling? fake user registration?)
- Many think that the "visible" part is by now only a minor portion (16-20%) of the entire web.

Problems

Which pages should be fetched?

Problems

Which pages should be fetched?

- Choosing the pages on the basis of their (supposed) content (header content-type)

Problems

Which pages should be fetched?

- Choosing the pages on the basis of their (supposed) content (header content-type)
- Only text/html? Or also other formats? (P.es., application/ms-word, application/postscript, . . .)

Problems

Which pages should be fetched?

- Choosing the pages on the basis of their (supposed) content (header content-type)
- Only text/html? Or also other formats? (P.es., application/ms-word, application/postscript, . . .)
- You need tools to parse other hypertextual formats (like PDF) different from HTML (to extract links, words etc.)

Problems

What information should you save for every page?

Problems

What information should you save for every page?

- Whole content (maybe: including HTTP headers)? Text only (no tags)? Only the sequence of word occurrences (no tags, no punctuation)?

Problems

What information should you save for every page?

- Whole content (maybe: including HTTP headers)? Text only (no tags)? Only the sequence of word occurrences (no tags, no punctuation)?
- Influences the amount of disk space

Problems

What information should you save for every page?

- Whole content (maybe: including HTTP headers)? Text only (no tags)? Only the sequence of word occurrences (no tags, no punctuation)?
- Influences the amount of disk space
- It is always necessary to keep the store in compressed form

Problems

What information should you save for every page?

- Whole content (maybe: including HTTP headers)? Text only (no tags)? Only the sequence of word occurrences (no tags, no punctuation)?
- Influences the amount of disk space
- It is always necessary to keep the store in compressed form
- Note: a compressed page, with headers, may occupy on the average about 4KB (the whole web would occupy about 16 TeraBytes)

Problems

Which links should you follow?

Problems

Which links should you follow?

- Sometimes you don't want to visit the whole web, but just a part of it (e.g., the Italian web)

Problems

Which links should you follow?

- Sometimes you don't want to visit the whole web, but just a part of it (e.g., the Italian web)
- You must decide some criterion (what is the Italian web, after all? .it? any page that appears to be written in Italian? and how can you decide this? and even then, how can you reach all such pages?)

Problems

Which links should you follow?

- Sometimes you don't want to visit the whole web, but just a part of it (e.g., the Italian web)
- You must decide some criterion (what is the Italian web, after all? .it? any page that appears to be written in Italian? and how can you decide this? and even then, how can you reach all such pages?)
- The criteria that can be implemented more easily decide which links should be followed on the basis of some (purely synctactic) condition

Problems

Politeness

Problems

Politeness

- You should avoid "bombing" a host with too many consecutive (let alone: parallel) requests

Problems

Politeness

- You should avoid “bombing” a host with too many consecutive (let alone: parallel) requests
- Respect black listing and the `robots.txt` (as well as any other robot-exclusion) protocol

Problems

Politeness

- You should avoid "bombing" a host with too many consecutive (let alone: parallel) requests
- Respect black listing and the `robots.txt` (as well as any other robot-exclusion) protocol
- While crawling, provide information (`user-agent`) that allow WebMasters to know who they should blame

Problems

Visit strategies

Problems

Visit strategies

- How do you select the next page to be visited from F (in the visit algorithm)?

Problems

Visit strategies

- How do you select the next page to be visited from F (in the visit algorithm)?
- Depth-first? breadth-first?

Problems

Visit strategies

- How do you select the next page to be visited from F (in the visit algorithm)?
- Depth-first? breadth-first?
- From this choice depends how fast you reach "important" (high-quality) pages

Problems

Visit strategies

- How do you select the next page to be visited from F (in the visit algorithm)?
- Depth-first? breadth-first?
- From this choice depends how fast you reach “important” (high-quality) pages
- Under many quality metrics, for general-purpose (non-focused) crawls, breadth-first visits seem to be the most efficient (but. . .)

Problems

How should one react to anomalies?

Problems

How should one react to anomalies?

- A crawler should be robust enough to tolerate anomalies (servers that don't respect the HTTP protocol; pages whose HTML syntax is not correct; etc.)

Problems

How should one react to anomalies?

- A crawler should be robust enough to tolerate anomalies (servers that don't respect the HTTP protocol; pages whose HTML syntax is not correct; etc.)
- Often, this need requires the usage of heuristics (e.g., if a server says it is going to serve a text/html page, should we trust it? how can we check if this is true?)

Problems

Snapshot vs. refresh

Problems

Snapshot vs. refresh

- The crawler structure we described is suitable only if you want to take a *snapshot* of the Web

Problems

Snapshot vs. refresh

- The crawler structure we described is suitable only if you want to take a *snapshot* of the Web
- But the Web keeps changing!

Problems

Snapshot vs. refresh

- The crawler structure we described is suitable only if you want to take a *snapshot* of the Web
- But the Web keeps changing!
- A crawler should continuously refresh its pages, visiting the same page over and over again

Problems

Snapshot vs. refresh

- The crawler structure we described is suitable only if you want to take a *snapshot* of the Web
- But the Web keeps changing!
- A crawler should continuously refresh its pages, visiting the same page over and over again
- One should respect the politeness policies while guaranteeing the maximal possible freshness

Distributed crawlers: why

A centralized crawler is extremely inefficient: most of the time, it is waiting for I/O.

Distributed crawlers: why

A centralized crawler is extremely inefficient: most of the time, it is waiting for I/O.

The obvious solution to this issue is to switch to a multi-process or distributed crawer.

Distributed crawlers: why

A centralized crawler is extremely inefficient: most of the time, it is waiting for I/O.

The obvious solution to this issue is to switch to a multi-process or distributed crawer. In many cases, practical crawlers are both:

Distributed crawlers: why

A centralized crawler is extremely inefficient: most of the time, it is waiting for I/O.

The obvious solution to this issue is to switch to a multi-process or distributed crawer. In many cases, practical crawlers are both:

- it is distributed, made by many *agents*, each performing a part of the crawl, i.e., visiting a portion of the URLs to be visited;

Distributed crawlers: why

A centralized crawler is extremely inefficient: most of the time, it is waiting for I/O.

The obvious solution to this issue is to switch to a multi-process or distributed crawer. In many cases, practical crawlers are both:

- it is distributed, made by many *agents*, each performing a part of the crawl, i.e., visiting a portion of the URLs to be visited;
- it is multi-process: within each agent, you have many processes performing parallel visits, or providing other functionalities (e.g., DNS, storage handling etc.).

Distributed crawlers: why

A centralized crawler is extremely inefficient: most of the time, it is waiting for I/O.

The obvious solution to this issue is to switch to a multi-process or distributed crawer. In many cases, practical crawlers are both:

- it is distributed, made by many *agents*, each performing a part of the crawl, i.e., visiting a portion of the URLs to be visited;
- it is multi-process: within each agent, you have many processes performing parallel visits, or providing other functionalities (e.g., DNS, storage handling etc.).

A distributed crawler features all the problems seen above, and some more. . .

Structure of a distributed crawler

Structure of a distributed crawler

- Every agent performs all the task of a usual crawer: it fetches a page, stores it somewhere, analyze its content looking for hyperlinks etc.

Structure of a distributed crawler

- Every agent performs all the task of a usual crawer: it fetches a page, stores it somewhere, analyze its content looking for hyperlinks etc.
- The agents run on some machines, that communicate with one another over a LAN (*intra-site parallel crawlers*) or on a geographical-sized network (*distributed crawler* in a proper sense).

Structure of a distributed crawler

- Every agent performs all the task of a usual crawer: it fetches a page, stores it somewhere, analyze its content looking for hyperlinks etc.
- The agents run on some machines, that communicate with one another over a LAN (*intra-site parallel crawlers*) or on a geographical-sized network (*distributed crawler* in a proper sense).
- You might have a central coordinator, that keeps track of the way the crawl is going on, or the agents may be losely coupled (in such case we speak of a *fully distributed crawler*).

Anarchic fully distributed crawlers

Anarchic fully distributed crawlers

- No central coordinator.

Anarchic fully distributed crawlers

- No central coordinator.
- Every agent starts from a certain seed and proceeds in a completely independent way, without any coordination with the other agents.

Anarchic fully distributed crawlers

- No central coordinator.
- Every agent starts from a certain seed and proceeds in a completely independent way, without any coordination with the other agents.
- **Pro's:**

Anarchic fully distributed crawlers

- No central coordinator.
- Every agent starts from a certain seed and proceeds in a completely independent way, without any coordination with the other agents.
- **Pro's:** No bottleneck,

Anarchic fully distributed crawlers

- No central coordinator.
- Every agent starts from a certain seed and proceeds in a completely independent way, without any coordination with the other agents.
- **Pro's:** No bottleneck, no single point of failure,

Anarchic fully distributed crawlers

- No central coordinator.
- Every agent starts from a certain seed and proceeds in a completely independent way, without any coordination with the other agents.
- **Pro's:** No bottleneck, no single point of failure, no inter-agent communication.

Anarchic fully distributed crawlers

- No central coordinator.
- Every agent starts from a certain seed and proceeds in a completely independent way, without any coordination with the other agents.
- **Pro's:** No bottleneck, no single point of failure, no inter-agent communication.
- **Con's:**

Anarchic fully distributed crawlers

- No central coordinator.
- Every agent starts from a certain seed and proceeds in a completely independent way, without any coordination with the other agents.
- **Pro's:** No bottleneck, no single point of failure, no inter-agent communication.
- **Con's:** Typically, many agents will end up crawling portions of Web that partially overlap (*overlap*).

Anarchic fully distributed crawlers

- No central coordinator.
- Every agent starts from a certain seed and proceeds in a completely independent way, without any coordination with the other agents.
- **Pro's:** No bottleneck, no single point of failure, no inter-agent communication.
- **Con's:** Typically, many agents will end up crawling portions of Web that partially overlap (*overlap*). Moreover, it is impossible to put in place reasonable politeness policies.

Fully distributed crawlers with static assignment

Fully distributed crawlers with static assignment

- No central coordinator.

Fully distributed crawlers with static assignment

- No central coordinator.
- The set U of URLs to be visited is partitioned *a-priori* in n subsets (one for each agent) $U_1, \ldots, U_n$.

Fully distributed crawlers with static assignment

- No central coordinator.
- The set U of URLs to be visited is partitioned *a-priori* in n subsets (one for each agent) $U_1, \ldots, U_n$.
- Every time a URL $u \in U_i$ is found, it is communicated to agent i that is responsible for it.

Fully distributed crawlers with static assignment

- No central coordinator.
- The set U of URLs to be visited is partitioned *a-priori* in n subsets (one for each agent) $U_1, \ldots, U_n$.
- Every time a URL $u \in U_i$ is found, it is communicated to agent i that is responsible for it.
- **Pro's:**

Fully distributed crawlers with static assignment

- No central coordinator.
- The set U of URLs to be visited is partitioned *a-priori* in n subsets (one for each agent) $U_1, \ldots, U_n$.
- Every time a URL $u \in U_i$ is found, it is communicated to agent i that is responsible for it.
- **Pro's:** No overlap;

Fully distributed crawlers with static assignment

- No central coordinator.
- The set U of URLs to be visited is partitioned *a-priori* in n subsets (one for each agent) $U_1, \ldots, U_n$.
- Every time a URL $u \in U_i$ is found, it is communicated to agent i that is responsible for it.
- **Pro's:** No overlap; if the partitioning respects the host part, it is possible to be polite.

Fully distributed crawlers with static assignment

- No central coordinator.
- The set U of URLs to be visited is partitioned *a-priori* in n subsets (one for each agent) $U_1, \ldots, U_n$.
- Every time a URL $u \in U_i$ is found, it is communicated to agent i that is responsible for it.
- **Pro's:** No overlap; if the partitioning respects the host part, it is possible to be polite.
- **Con's:**

Fully distributed crawlers with static assignment

- No central coordinator.
- The set U of URLs to be visited is partitioned *a-priori* in n subsets (one for each agent) $U_1, \ldots, U_n$.
- Every time a URL $u \in U_i$ is found, it is communicated to agent i that is responsible for it.
- **Pro's:** No overlap; if the partitioning respects the host part, it is possible to be polite.
- **Con's:** If an agent stops working, the entire portion of the Web it was assigned gets lost;

Fully distributed crawlers with static assignment

- No central coordinator.
- The set U of URLs to be visited is partitioned *a-priori* in n subsets (one for each agent) $U_1, \ldots, U_n$.
- Every time a URL $u \in U_i$ is found, it is communicated to agent i that is responsible for it.
- **Pro's:** No overlap; if the partitioning respects the host part, it is possible to be polite.
- **Con's:** If an agent stops working, the entire portion of the Web it was assigned gets lost; it is impossible to change the set of working agents while the crawler is running: you have to start from scratch;

Fully distributed crawlers with static assignment

- No central coordinator.
- The set U of URLs to be visited is partitioned *a-priori* in n subsets (one for each agent) $U_1, \ldots, U_n$.
- Every time a URL $u \in U_i$ is found, it is communicated to agent i that is responsible for it.
- **Pro's:** No overlap; if the partitioning respects the host part, it is possible to be polite.
- **Con's:** If an agent stops working, the entire portion of the Web it was assigned gets lost; it is impossible to change the set of working agents while the crawler is running: you have to start from scratch; it is impossible to set-up a reasonable load-balancing between agents (you can just hope that balancing the cardinalities implies balancing workload);

Fully distributed crawlers with static assignment

- No central coordinator.
- The set U of URLs to be visited is partitioned *a-priori* in n subsets (one for each agent) $U_1, \ldots, U_n$.
- Every time a URL $u \in U_i$ is found, it is communicated to agent i that is responsible for it.
- **Pro's:** No overlap; if the partitioning respects the host part, it is possible to be polite.
- **Con's:** If an agent stops working, the entire portion of the Web it was assigned gets lost; it is impossible to change the set of working agents while the crawler is running: you have to start from scratch; it is impossible to set-up a reasonable load-balancing between agents (you can just hope that balancing the cardinalities implies balancing workload); a lot of communication.

Crawlers with central coordinator

Crawlers with central coordinator

- The central coordinator keeps track of which URLs have already been visited, and at every new URL it decides the agent that will fetch the page, usually on the basis of the current agents' workload.

Crawlers with central coordinator

- The central coordinator keeps track of which URLs have already been visited, and at every new URL it decides the agent that will fetch the page, usually on the basis of the current agents' workload.
- Every agent, after fetching a page, must communicate the outlinks to the central coordinator.

Crawlers with central coordinator

- The central coordinator keeps track of which URLs have already been visited, and at every new URL it decides the agent that will fetch the page, usually on the basis of the current agents' workload.
- Every agent, after fetching a page, must communicate the outlinks to the central coordinator.
- **Pro's:**

Crawlers with central coordinator

- The central coordinator keeps track of which URLs have already been visited, and at every new URL it decides the agent that will fetch the page, usually on the basis of the current agents' workload.
- Every agent, after fetching a page, must communicate the outlinks to the central coordinator.
- **Pro's:** No overlap;

Crawlers with central coordinator

- The central coordinator keeps track of which URLs have already been visited, and at every new URL it decides the agent that will fetch the page, usually on the basis of the current agents' workload.
- Every agent, after fetching a page, must communicate the outlinks to the central coordinator.
- **Pro's:** No overlap; (potentially) optimal load-balancing;

Crawlers with central coordinator

- The central coordinator keeps track of which URLs have already been visited, and at every new URL it decides the agent that will fetch the page, usually on the basis of the current agents' workload.
- Every agent, after fetching a page, must communicate the outlinks to the central coordinator.
- **Pro's:** No overlap; (potentially) optimal load-balancing; the number of agent may change during the crawl.

Crawlers with central coordinator

- The central coordinator keeps track of which URLs have already been visited, and at every new URL it decides the agent that will fetch the page, usually on the basis of the current agents' workload.
- Every agent, after fetching a page, must communicate the outlinks to the central coordinator.
- **Pro's:** No overlap; (potentially) optimal load-balancing; the number of agent may change during the crawl.
- **Con's:**

Crawlers with central coordinator

- The central coordinator keeps track of which URLs have already been visited, and at every new URL it decides the agent that will fetch the page, usually on the basis of the current agents' workload.
- Every agent, after fetching a page, must communicate the outlinks to the central coordinator.
- **Pro's:** No overlap; (potentially) optimal load-balancing; the number of agent may change during the crawl.
- **Con's:** It requires an enormous amount of information exchange;

Crawlers with central coordinator

- The central coordinator keeps track of which URLs have already been visited, and at every new URL it decides the agent that will fetch the page, usually on the basis of the current agents' workload.
- Every agent, after fetching a page, must communicate the outlinks to the central coordinator.
- **Pro's:** No overlap; (potentially) optimal load-balancing; the number of agent may change during the crawl.
- **Con's:** It requires an enormous amount of information exchange; the central coordinator is both a communication bottleneck and a single point of failure.

Fully distributed crawlers with dynamic assignment

Fully distributed crawlers with dynamic assignment

- Agents' responsibilities are determined on the bases of a *responsibility function*.

Fully distributed crawlers with dynamic assignment

- Agents' responsibilities are determined on the bases of a *responsibility function*.
- This function determines, for every URL, the agent responsible for it, in such a way that:

Fully distributed crawlers with dynamic assignment

- Agents' responsibilities are determined on the bases of a *responsibility function*.
- This function determines, for every URL, the agent responsible for it, in such a way that:
  - every agent can autonomously (=w/o communication) establish who is responsible for a given URL;

Fully distributed crawlers with dynamic assignment

- Agents' responsibilities are determined on the bases of a *responsibility function*.
- This function determines, for every URL, the agent responsible for it, in such a way that:
  - every agent can autonomously (=w/o communication) establish who is responsible for a given URL;
  - all agents agree on their decisions;

Fully distributed crawlers with dynamic assignment

- Agents' responsibilities are determined on the bases of a *responsibility function*.
- This function determines, for every URL, the agent responsible for it, in such a way that:
  - every agent can autonomously (=w/o communication) establish who is responsible for a given URL;
  - all agents agree on their decisions;
  - if a new agent is added, it will take some of the responsibilities of the existing agents, but this will happen without causing conflicts in the responsibilities of the agents themselves.

More precisely. . .

Let A be the set of *potential agents*, and U be the set of URLs. In every moment, there will be a set $X \subseteq A$ of *alive agents*; given this set, you must decide, for each URL, who (among the alive agent) is responsible for that URL.

More precisely. . .

The exact requirements follow:

More precisely...

The exact requirements follow:

- There is a responsibility function $r : 2^A \times U \to A$

More precisely...

The exact requirements follow:

- There is a responsibility function $r : 2^A \times U \to A$
- For every $\emptyset \neq X \subseteq A$ and $u \in U$ you must have $r(X, u) \in X$ (*correctness*)

More precisely. . .

The exact requirements follow:

- There is a responsibility function $r : 2^A \times U \to A$
- For every $\emptyset \neq X \subseteq A$ and $u \in U$ you must have $r(X, u) \in X$ (*correctness*)
- If $x \in X$, the set $\{u \in U \mid r(X, u) = x\}$ must have cardinality $\approx |U|/|X|$ (*equity*)

More precisely...

The exact requirements follow:

- There is a responsibility function $r : 2^A \times U \to A$
- For every $\emptyset \neq X \subseteq A$ and $u \in U$ you must have $r(X, u) \in X$ (*correctness*)
- If $x \in X$, the set $\{u \in U \mid r(X, u) = x\}$ must have cardinality $\approx |U|/|X|$ (*equity*)
- Suppose $y \notin X$: if $r(X \cup \{y\}, u) = x \in X$ then $r(X, u) = x$ (*covariance*);

More precisely...

The exact requirements follow:

- There is a responsibility function $r : 2^A \times U \to A$
- For every $\emptyset \neq X \subseteq A$ and $u \in U$ you must have $r(X, u) \in X$ (*correctness*)
- If $x \in X$, the set $\{u \in U \mid r(X, u) = x\}$ must have cardinality $\approx |U|/|X|$ (*equity*)
- Suppose $y \notin X$: if $r(X \cup \{y\}, u) = x \in X$ then $r(X, u) = x$ (*covariance*); in other words: adding a new agent y cannot change the responsibilities of existing agents, except for the fact that some responsibilities will of course be given to y.

You can implement this. . .

. . . by using a consistent hashing function. Typically:

You can implement this…

…by using a consistent hashing function. Typically:

- you map every agent $a \in A$ to a random set (seeded on the agent's identity) of K points on the unit circle (called *replicas*)

You can implement this. . .

. . . by using a consistent hashing function. Typically:

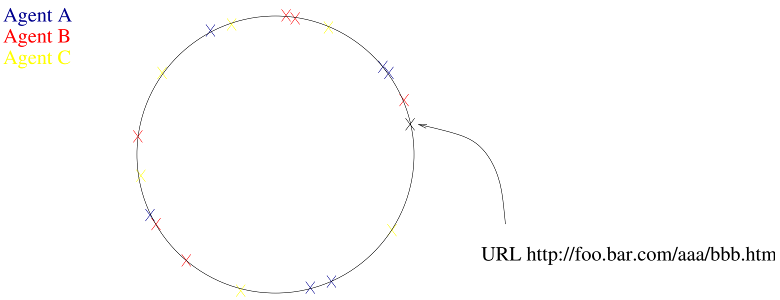
- you map every agent $a \in A$ to a random set (seeded on the agent's identity) of K points on the unit circle (called *replicas*)
- you similarly map every URL $u \in U$ to a random point (seeded on the URL, or the URL's host part)

You can implement this. . .

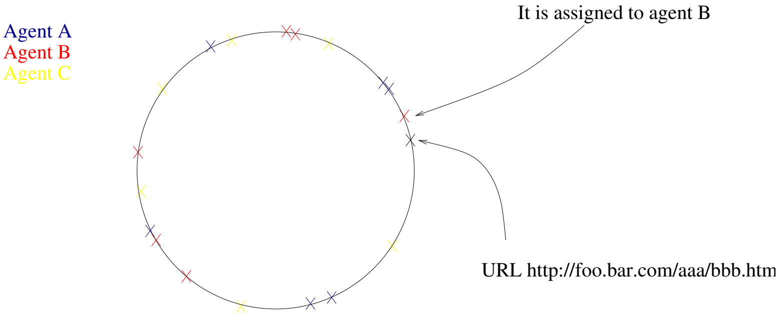
. . . by using a consistent hashing function. Typically:

- you map every agent $a \in A$ to a random set (seeded on the agent's identity) of K points on the unit circle (called *replicas*)
- you similarly map every URL $u \in U$ to a random point (seeded on the URL, or the URL's host part)
- $r(X, u)$ is found by determining the replica of an alive agent that is closest to the point assigned to u.

Consistent hashing



Consistent hashing



Consistent hashing