Condor e DAGMan

Napoli 2007

Condor Project
Computer Sciences Department
University of Wisconsin-Madison
condor-admin@cs.wisc.edu
http://www.cs.wisc.edu/condor

Alcuni job hanno dipendenze…

I risultati di un job, possono servire come dati di input di un altro job…

I DAGMan

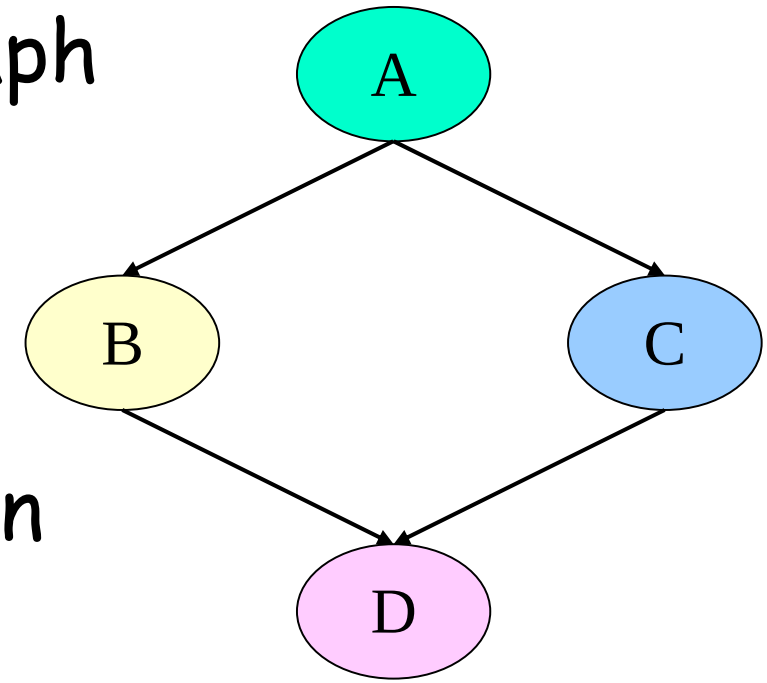
- **Directed Acyclic Graph Manager**
- Il DAGMan consente di specificare le dipendenze tra i job, in modo che Condor li gestisca in modo automatico.
- Un esempio di dipendenza: non eseguire il job B fino a quando il job A non è stato completato con successo.

Cos'è un DAG?

- Directed Acyclic Graph
- Un grafo aciclico orientato è una

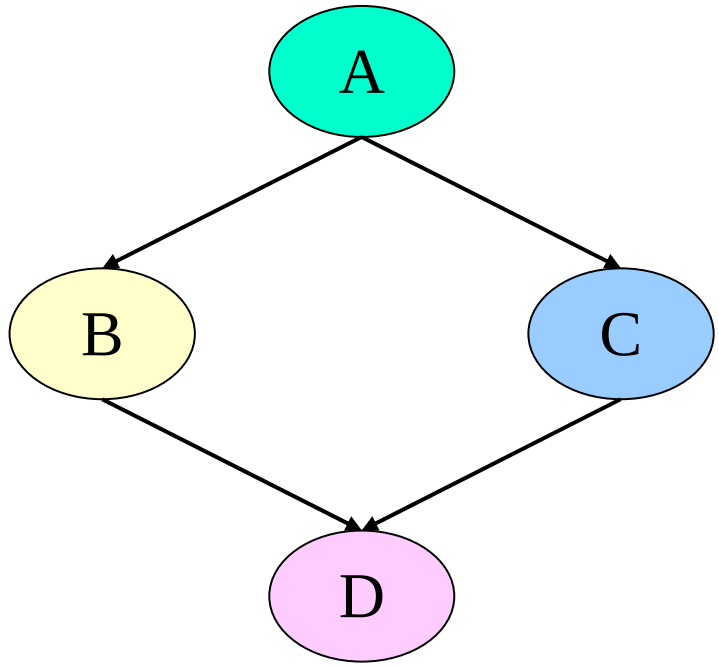
  struttura dati

  utilizzata da DAGMan per rappresentare le dipendenze



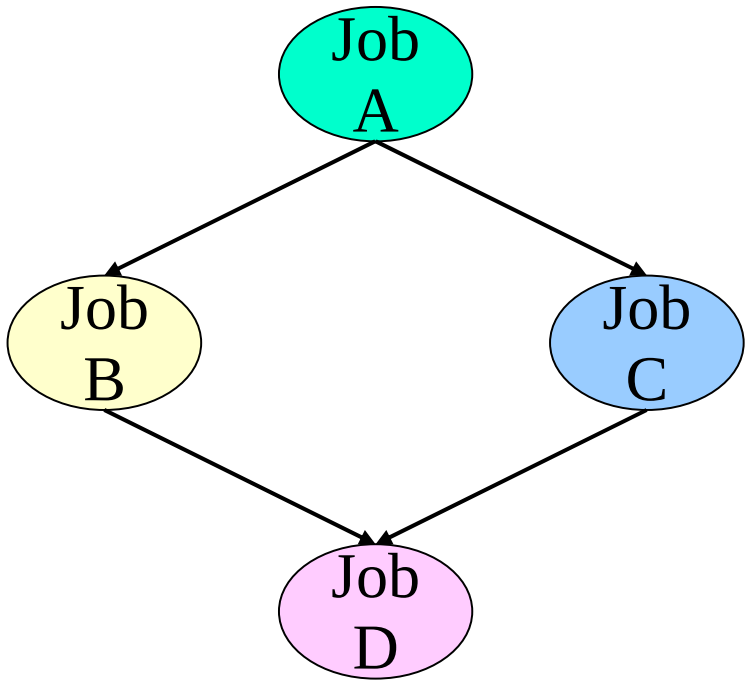
Definizione di DAG

- I DAG hanno uno o più nodi (o vertici).
- Le dipendenze sono rappresentate da archi (o lati): frecce che vanno da un nodo padre ad un nodo figlio.
- Non ci sono cicli !



Condor e i DAG

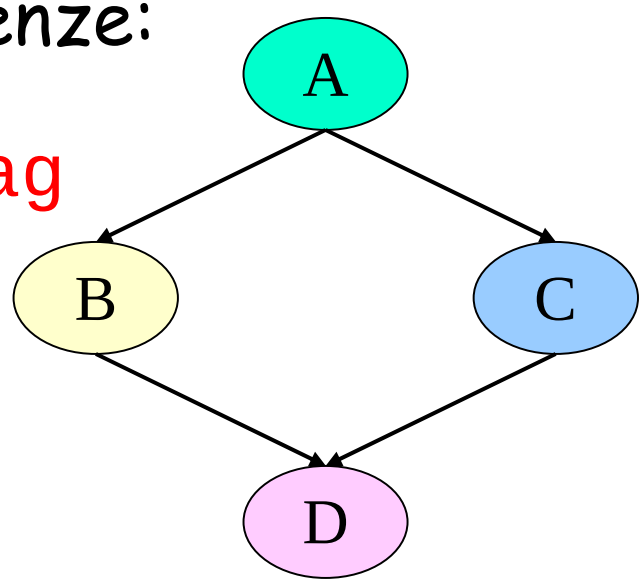
- Ogni nodo rappresenta un job di Condor
- Le dipendenze definiscono il possibile ordine di esecuzione del job



Definire un DAG per l'esecuzione in Condor

Il DAG viene definito attraverso un file testo che definisce tutte le dipendenze:

```
# file name: diamond.dag
Job A a.submit
Job B b.submit
Job C c.submit
Job D d.submit
Parent A Child B C
Parent B C Child D
```



Submit Description File

Per il nodo A:

```
# file name:
#    a.submit
universe   = vanilla
executable = A
input      = A.in
output     = A.out
error      = A.err
log        = A.log
queue
```

Per il nodo B:

```
# file name:
#    b.submit
universe   = vanilla
executable = B
input      = B.in
output     = B.out
error      = B.err
log        = B.log
queue
```

Per il nodo C:

```
# file name:
#    c.submit
universe   = standard
executable = C
input      = C.in
output     = C.out
error      = C.err
log        = C.log
queue
```

Per il nodo D:

```
# file name:
#    d.submit
universe   = standard
executable = D
input      = D.in
output     = D.out
error      = D.err
log        = D.log
queue
```

Sottomettere il DAG a Condor

- Per sottomettere il DAG si esegue il comando:

```
condor_submit_dag diamond.dag
```

- condor_submit_dag crea un submit description file per DAGMan, e il DAG viene sottomesso come job di Condor!

Un requisito di DAGMan

- Nel submit description file di ogni job del DAG deve essere specificato un file di log
- I file di log possono essere separati o condivisi da più job dello stesso DAG
- I file di log sono utilizzati utilizzati per sincronizzare la sottomissione dei job del DAG

Nodi

- L'esecuzione di un Job corrispondente ad un nodo può

  Terminare correttamente

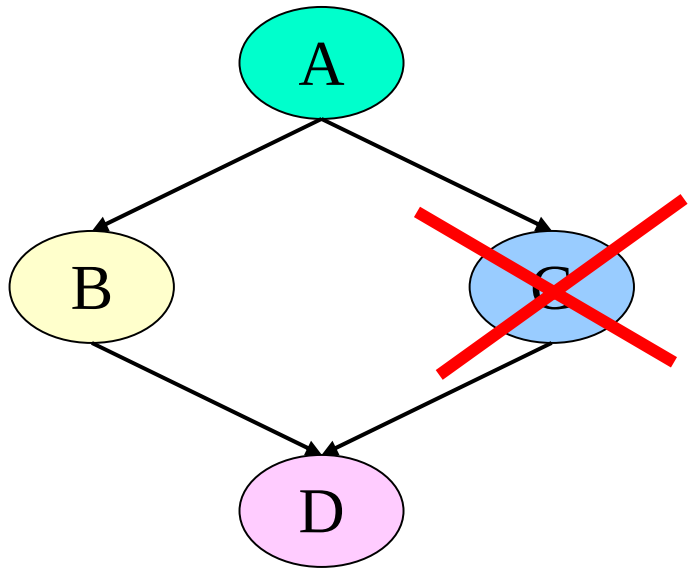
  o

  fallire

- Condor si basa sul valore di ritorno del job

  0 : successo

  non 0: fallimento



Uso avanzato di DAGMan

- Ripetere l'esecuzione di un nodo
- Terminare anticipatamente (abort) un intero DAG
- Utilizzo delle variabili VARS
- Utilizzo dei Throttle
- PRE e POST script nell'editing di un DAG
- DAG innestati

Ripetizione di un nodo nel DAG

- Prima di considerare un nodo fallito Condor può:
  - Ripetere N volte la sua esecuzione se nell'input file del DAG si specifica:

    ```
    Retry C 4
    ```

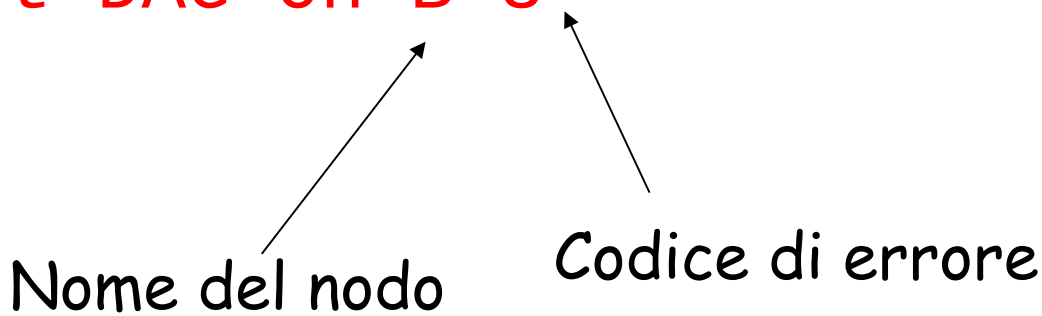
    (ripetere l'esecuzione di C, 4 volte prima di considerare il nodo fallito)
  - Ripetere N volte la sua esecuzione, a meno che l'eseguibile non ritorni uno specifico codice di uscita, se nell'input file del DAG si specifica:

    ```
    Retry C 4 UNLESS-EXIT 2
    ```

Interrompere l'esecuzione di un DAG

- Quando uno specifico valore d'errore deve arrestare l'esecuzione dell'intero DAG
- Nell' input file del DAG si scriverà:

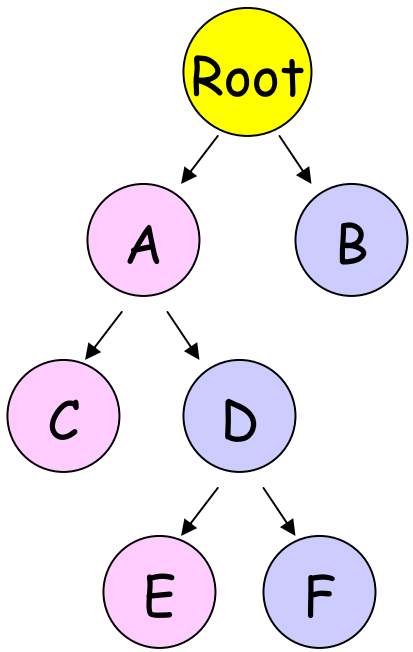
```
Abort-DAG-On B 3
```



VARS

- E' un elemento del file di input del DAG che consente di ridurre il numero di submit description file necessari da elencare
  - definisce una variabile ed un valore
  - si associa ad un nodo
  - sostituisce il valore in una macro

Esempio:
Un albero binario

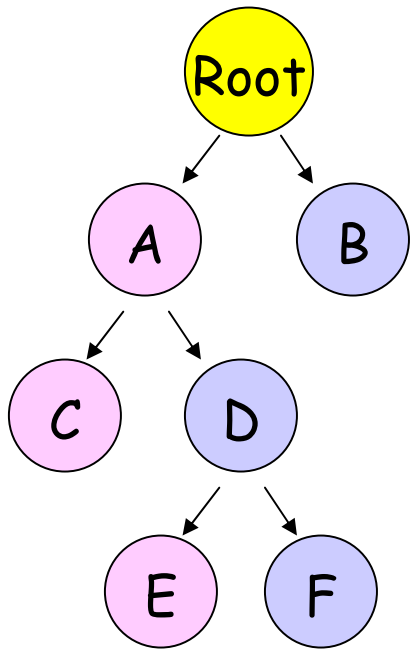


Si assuma che l'eseguibile
è lo stesso per ogni nodo

Ma si comporta
differentemente a seconda se
è un figlio destro o sinistro

L'input file del DAG TREE con VARS

```
# tree example, file is tree.dag
Job root node.submit
Job A node.submit
Vars A position="left"
Job B node.submit
Vars B position="right"
Job C node.submit
Vars C position="left"
. . .
Parent root Child A B
. . .
```



Il Submit Description File di ogni nodo DAG TREE

```
# file name is node.submit
executable = process.exe
arguments  = $(position)
log        = node.log
queue
```

Il job del nodo *A* ha linea di commando:

```
process.exe left
```

Throttle

- I Throttle controllano il numero di sottomissioni di job
  - Massimo numero di job sottomessi

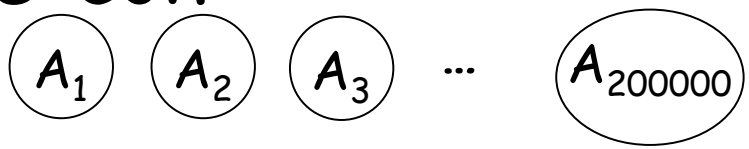
```
% condor_submit_dag -maxjobs 40 bigdag.dag
```

  - Massimo numero di job idle

```
% condor_submit_dag -maxidle 10 bigdag.dag
```

Un esempio di Throttling

- Sottomettiamo un DAG con
  - 200,000 nodi
  - Nessuna dipendenza tra job



- Usare DAGMan per rallentare (throttle) i job, ma…

Nonostante Condor sia scalabile, l'esecuzione di 200,000 job simultanei potrebbe generare problemi

script DAGMan

- DAGMan consente di eseguire PRE e/o POST script

(Non deve essere necessariamente uno script, può essere utilizzato qualsiasi eseguibile)

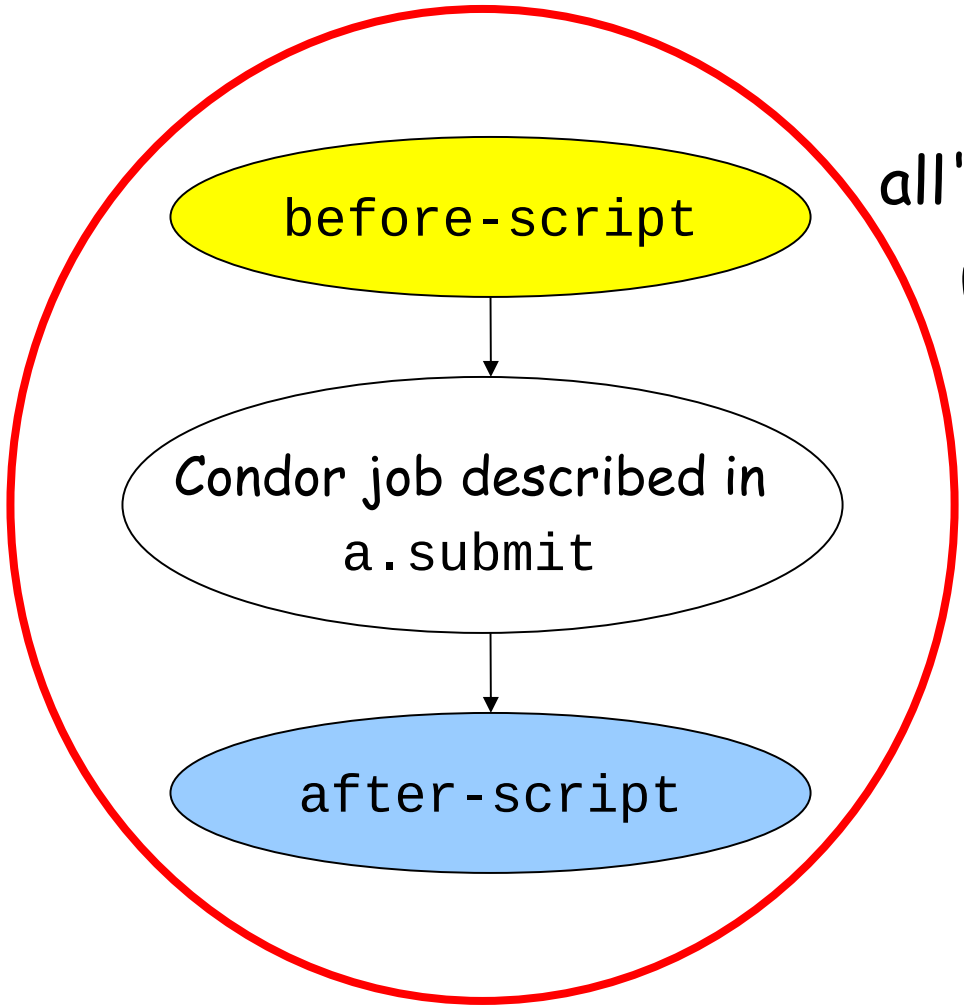
Quando?

- prima (PRE) o dopo (POST) il job

Dove?

- sull'host che ha sottomesso il job
- Da specificare nel file di input del DAG:

```
Job A a.submit
Script PRE A before-script <arguments>
Script POST A after-script <arguments>
```



nodo A
all'interno di
un DAG

script PRE

Gli script PRE possono prendere decisioni

- Devo passare argomenti differenti al job?
- Devo cambiare il submit description file?

script POST

- Gli script POST vengono sempre eseguiti indipendentemente dal valore di ritorno del Job
- Gli script POST possono cambiare il valore di ritorno del nodo
  - DAGMan considera il nodo fallito quando lo script POST restituisce un valore non nullo
  - Lo script POST può intepretare i codici restituiti dall'eseguibile e i file di output e restituire 0 (successo) o un valore non nullo (fallimento), in base a considerazioni decise dell'utente.

Variabili Pre-definite

- Nel file di input del DAG:

```
Job A a.submit
Script PRE A before-script $JOB
Script POST A after-script $JOB $RETURN
```

argomenti (opzionali) per lo script

$JOB contiene la stringa che definsce il nome del nodo

$RETURN contiene il valore restituito dal Condor job relativo al nodo

Il Throttle per gli script

- Il meccanismo del Throttle può controllare anche il numero di script che vengono eseguiti contemporaneamente

```
% condor_submit_dag –maxpre 10 bigdag.dag
```

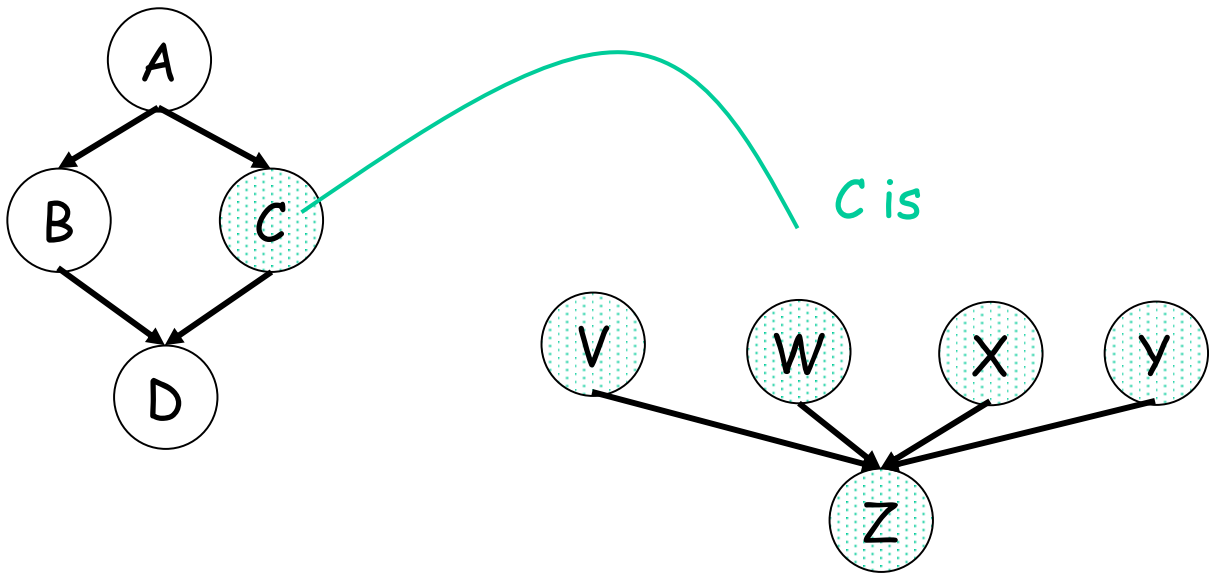
oppure

```
% condor_submit_dag –maxpost 30 bigdag.dag
```

DAG Innestati

- Ogni nodo di un DAG può essere considerato uno script che:
  1. Effettua una decisione
  2. Esegue un nuovo DAG
  3. Il nodo del DAG non sarà completato fino a che il DAG interno (innestato) non sarà terminato
- Quali sono i vantaggi?
  - Implementare un loop di lunghezza finita
  - Modificare il comportamento durante l'esecuzione

Esempio di DAG Innestati



Using Stork

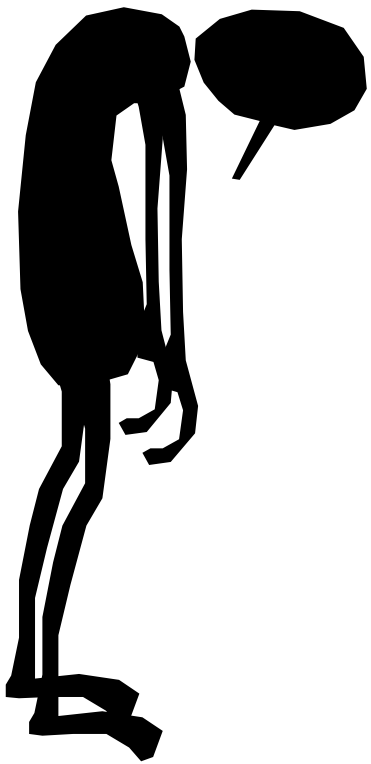
Napoli, 2007

Condor Project
Computer Sciences Department
University of Wisconsin-Madison
condor-admin@cs.wisc.edu
http://www.cs.wisc.edu/condor

Incontriamo Friedrich*

Friedrich è uno scenziato con un GRANDE problema.

*Il fratello gemello di Frieda



Ha un' *ingente quantità di dati* da processare.

Il problema di Friedrich

Friedrich ha molti insiemi di dati ingenti da processare. Per ogni insieme deve:

1. scaricare i dati da un server remoto
2. eseguire un job che processa i dati
3. salvare i dati di output su un server remoto

L'approccio classico al trasferimento dati

```
#!/bin/sh
globus-url-copy  source  dest
```

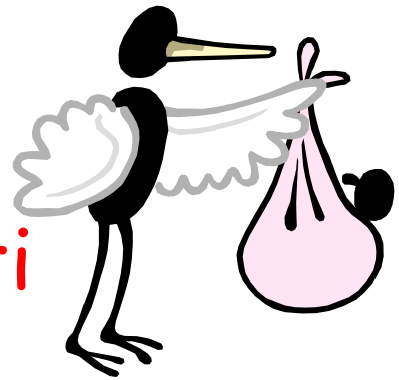
Gli script solitamente funzionano bene per trasferimenti di dati brevi e semplici, ma …

Le cose possono andare storte!!

Questi errori sono frequenti quando si manipolano ingenti quantità di dati:

- La connessione di rete cade.
- Il server non è disponibile.
- I dati trasferiti risultano corrotti.
- Il workflow non si accorge che I dati sono corrotti.

Stork Risolve Problemi



- Crea il concetto di

job per il posizionamento di dati

- Viene gestito e schedulato come qualsiasi altro job di Condor
- I job di Friedrich beneficiano della fault-tolerance

Protocolli di trasferimento dati supportati

- file system locale
- GridFTP
- FTP
- HTTP
- SRB
- NeST
- SRM
- è estendibile ad altri protocolli

Fault Tolerance

- Riprova ad eseguire job falliti
- Nei tentativi successivi può utilizzare un protocollo alternativo
  - Per esempio, prima prova GridFTP, poi prova FTP
- Riprova trasferimenti bloccati
- Reazioni ai fallimenti configurabili

Getting Stork

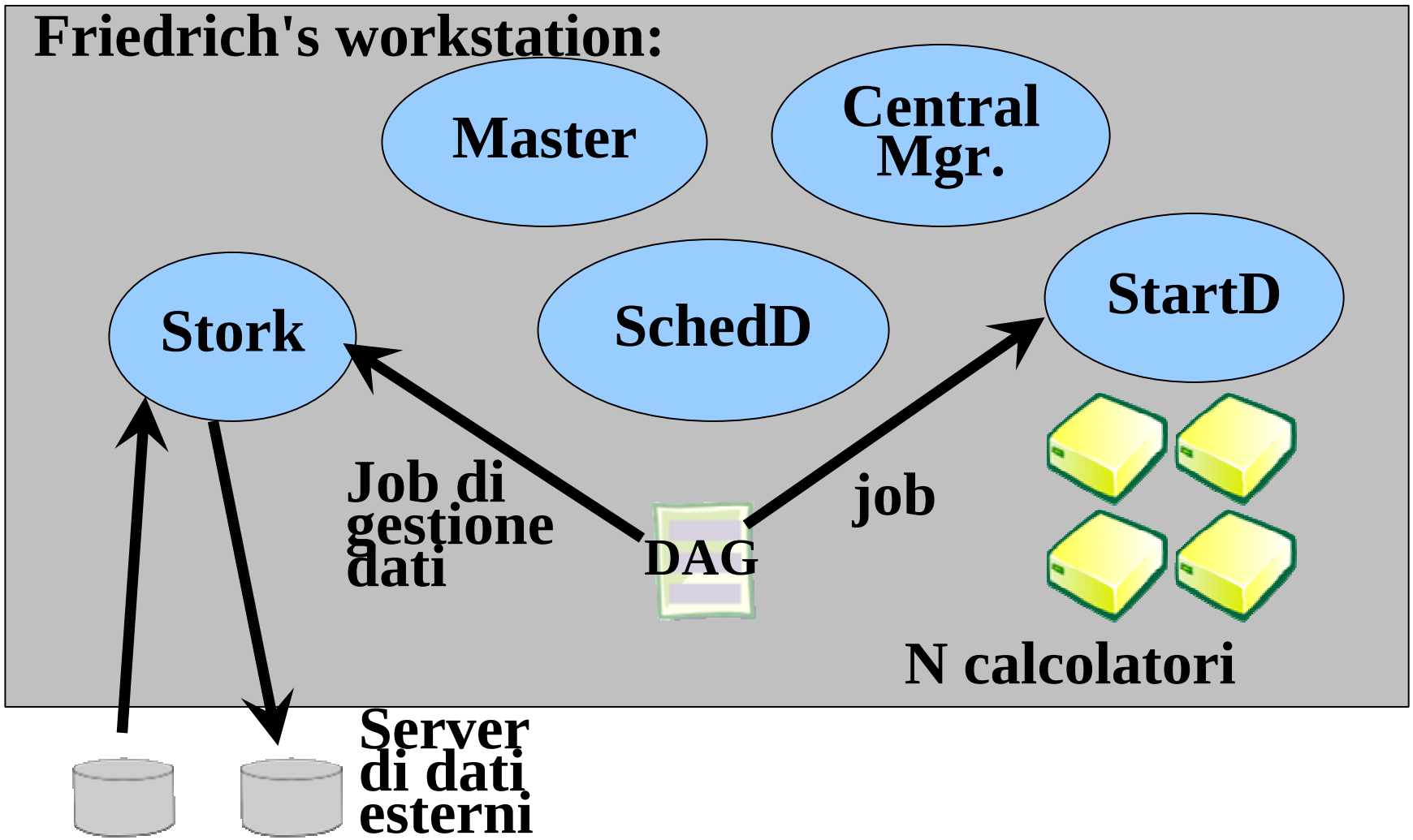
- Stork is part of Condor, so get Condor. . .
- Available as a free download from

  http://www.cs.wisc.edu/condor
- Currently available for Linux platforms

Stork funziona bene anche con il Personal Condor

- L'installazione di Condor/Stork sulla propria workstation, non richiede privilegi di amministratore
- Dopo l'installazione Friedrich sottomente i suoi job al suo Personal Stork…

Personal Condor di Friedrich



Stork potrà…

- Controllare I job per la gestione dei dati consentendoti di monitorare come procedono
- Limitare il massimo numero di job in esecuzione
- Tenere un log delle attività dei job
- Aggiungere fault-tolerance a tutti i job
  - Individua e riprova i job che falliscono

Il Submit Description File

- Come per gli altri di Condor, un file di testo con un formato differente
- Scritto come nuova ClassAd
- Come condor non importa l'estensione del nome del file
- Il contenuto del file istruisce Stork sul job:
  - il tipo di movimento, sorgente/destinazione locazione/protocollo, impostazioni del proxy, protocolli alternativi da provare

Un semplice Submit File

```
// c++ style comment lines
// file name is stage-in.stork
[
  dap_type  = "transfer";
  src_url   = "http://server/path";
  dest_url  = "file:///dir/file";
  log       = "stage-in.log";
]
```

Nota: il formato è differente dal quello adottato da Condor

Un altro semplice Submit File

```
// sintassi c++ per commenti
// il nome del file è stage-in.stork
[
  dap_type  = "transfer";
  src_url   = “gsiftp://server/path";
  dest_url  = "file:///dir/file";
  x509proxy = "default";
  log       = "stage-in.log";
]
```

Nota: il formato è differente dal quello adottato da Condor

Eseguire stork_submit

- Diamo a **stork_submit** il nome del submit file:

```
% stork_submit  stage-in.stork
```

- **stork_submit** verifica il submit file, controlla se ci sono errori, ed invia il job al server Stork.
- **stork_submit** restituisce il job id (un riferimento univoco al job)

Sample stork_submit

```
% stork_submit stage-in.stork
================
Sending request:
    [
        dest_url = "file:///dir/file";
        src_url = “http://server/path";
        dap_type = "transfer";
        log = "path/stage-in.log";
    ]
================

Request assigned id: 1
```

job id

La coda di Job

- **`stork_submit`** invia il job al server Stork
  - Il server Stork gestisce la coda di job locale
- Visualizzare la coda con **`stork_q`**, o con **`stork_status`**

Lo stato del Job

- **stork_q** chiede informazioni sui job attivi

```
% stork_q
```

- **stork_status** chiede informazioni su un dato job attivo o completato

```
% stork_status 12
```

Eliminare job

- Per eliminare un job di posizionamento dati dalla coda si utilizza **stork_rm**
- Si possono soltanto rimuovere job propri (root può rimuovere i job di tutti)
- E' necessario specificare un job ID

```
% stork_rm 21
```

removes un job

Usare File di Log

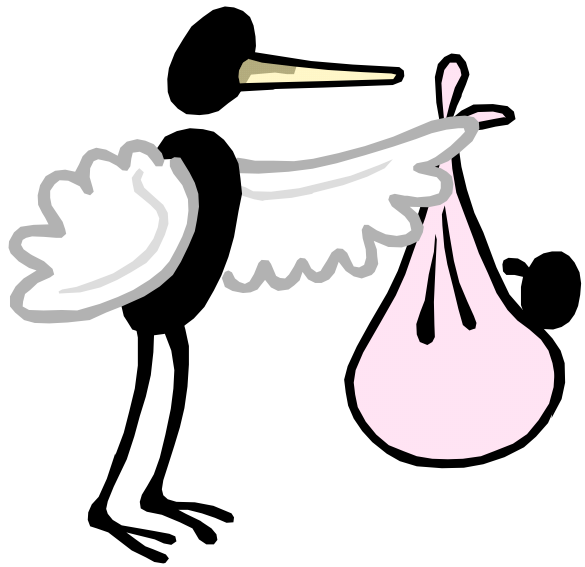
```
// sintassi c++ per commenti
[
  dap_type  = "transfer";
  src_url   = "gsiftp://server/path";
  dest_url  = "file:///dir/file";
  x509proxy = "default";
  log       = "stage-in.log";
]
```

Sample Stork User Log

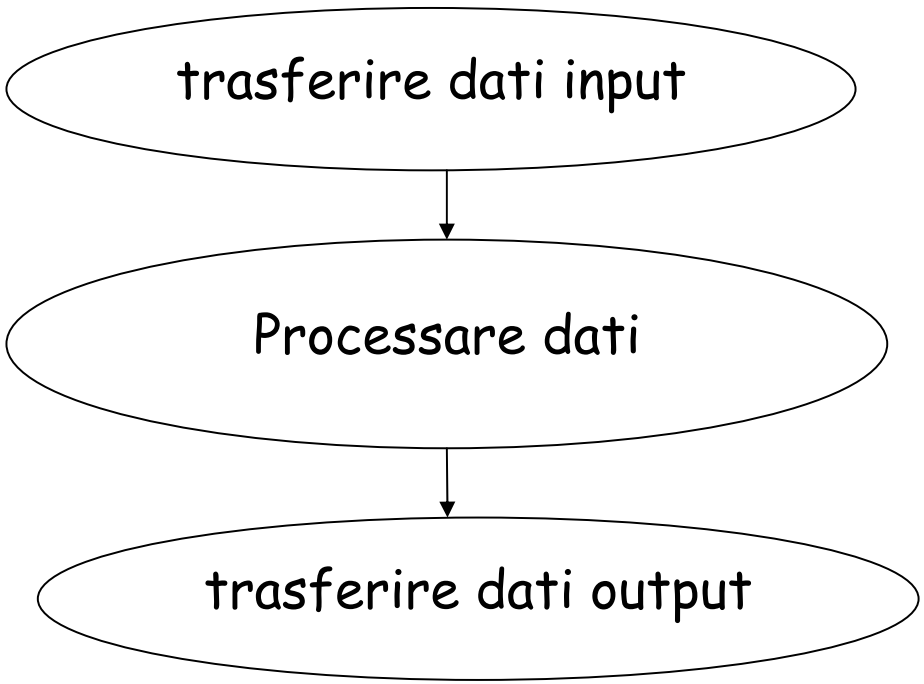
```
000 (001.-01.-01) 04/17 19:30:00 Job submitted from host: <128.105.121
...
001 (001.-01.-01) 04/17 19:30:01 Job executing on host: <128.105.121.5
...
008 (001.-01.-01) 04/17 19:30:01 job type: transfer
...
008 (001.-01.-01) 04/17 19:30:01 src_url: gsiftp://server/path
...
008 (001.-01.-01) 04/17 19:30:01 dest_url: file:///dir/file
...
005 (001.-01.-01) 04/17 19:30:02 Job terminated.
    (1) Normal termination (return value 0)
        Usr 0 00:00:00, Sys 0 00:00:00  -  Run Remote Usage
        Usr 0 00:00:00, Sys 0 00:00:00  -  Run Local Usage
        Usr 0 00:00:00, Sys 0 00:00:00  -  Total Remote Usage
        Usr 0 00:00:00, Sys 0 00:00:00  -  Total Local Usage
    0  -  Run Bytes Sent By Job
    0  -  Run Bytes Received By Job
    0  -  Total Bytes Sent By Job
    0  -  Total Bytes Received By Job
...
```

Stork e DAGMan

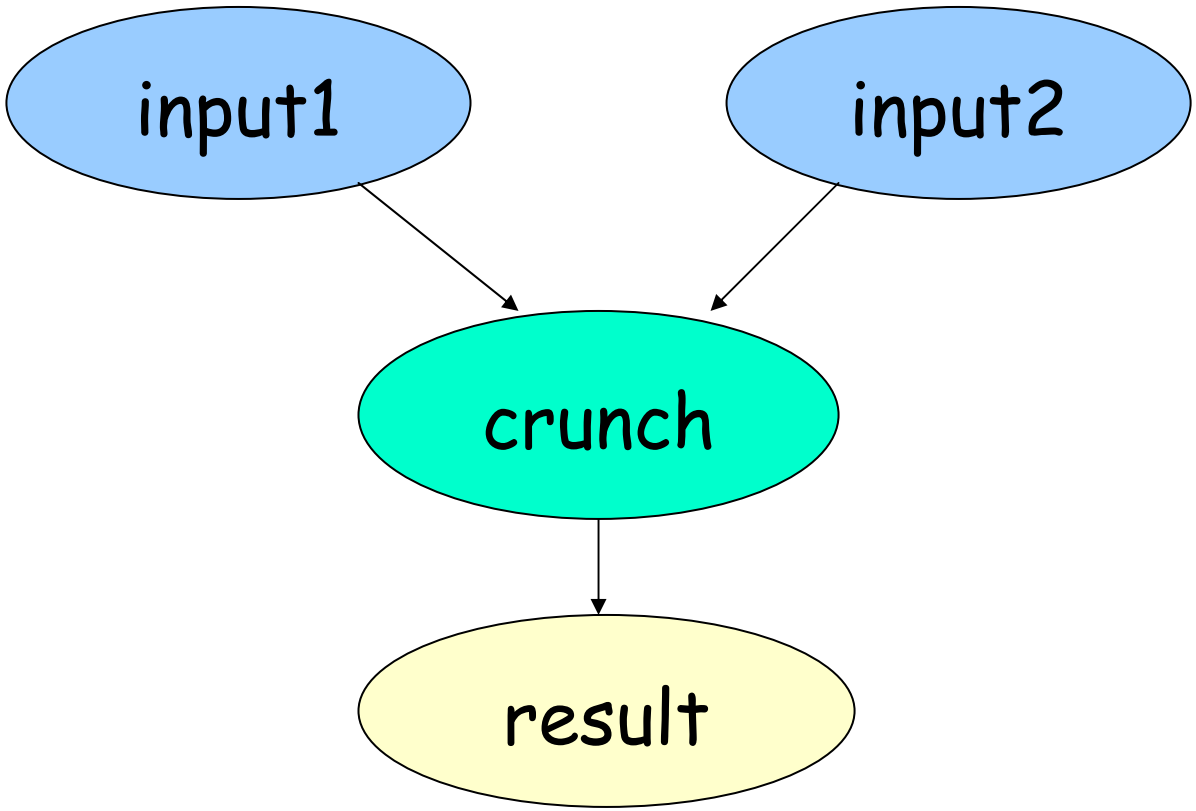
I job stork per il trasferimento dati sono integrati con DAGMan per la felicità di Friedrich



Diefiniamo il DAG di Friedrich



Il DAG di Friedrich



The DAG Input File

```
# il nome del file è friedrich.dag
DATA input1 input1.stork
DATA input2 input2.stork
JOB crunch process.submit
DATA result result.stork
PARENT input1 input2 CHILD crunch
PARENT crunch CHILD result
```

Un Submit File di Stork

```
// il nome del file è input1.stork
[
  dap_type  = "transfer";
  src_url   = “http://north.cs.wisc.edu/
     ~freidrich/data1";
  dest_url  = "file:///home/friedrich/in1";
  log       = "in1.log";
]
```

Il Submit Description File

```
# il nome del file è process.submit
universe    = vanilla
executable  = process
input       = in1
output      = crunch.result
error       = crunch.err
log         = crunch.log
queue
```

Stork Submit File

```
// file name is result.stork
[
  dap_type  = "transfer";
  src_url   =
        "file:///home/friedrich/crunch.result";
  dest_url  = “http://north.cs.wisc.edu/
        ~friedrich/final.results";
  log       = "result.log";
]
```

Friedrich sottomette il DAG

- Utilizzando come working directory `/home/friedrich`

```
% condor_submit_dag friedrich.dag
```

In pratica

Con Stork Friedrich ora può…

- Sottomettere un job per il processamento di dati e andare a casa! Perchè
  - Stork gestisce il trasferimento dati individuando eventuali malfunzionamenti
- DAGMan gestisce le dipendenze