I117（17）テキストファイル（4）httpd ログ

知念

北陸先端科学技術大学院大学 情報科学研究科
School of Information Science,
Japan Advanced Institute of Science and Technology

httpd ログ

- HTTP サーバ （httpd）では通信履歴を記録
- 記録はログファイル（log file）と呼ぶ
  - ファイル名は access_log や access.log
- 広く使われているのは CERN httpd 由来の common format
  - 今回は lighttpd のログ
    - common format と2つの拡張フィールド
  - 1000文字を越える長い行がある
    - 最大3108文字

httpd ログ *(cont.)*

```
123.45.67.78 localhost - [08/Jun/2008
:10:42:00 +0900] "GET /webplayer/index
.py HTTP/1.1" 200 15400 "-" "Mozilla/5
.0 (Macintosh; U; Intel Mac OS X; ja-J
P-mac; rv:1.8.1.14) Gecko/20080404 Fir
efox/2.0.0.14"
```

全体の区切り文字は空白

- 時刻はブランケット
- いくつかダブルクォート

0	**相手 IPアドレス**	**123.45.67.89**
1	**ログネーム**	**localhost**
2	**認証ユーザネーム**	**-**
3	**時刻**	**[08/Jun/2008:10:42:00 +0900]**
4	**リクエスト**	**”GET /webplayer/index.py HTTP/1.1”**
5	**ステータスコード**	**200**
6	**転送バイト数**	**15400**
7	**Referer**	**”-”**
8	**User-Agent**	**”Mozilla/5.0 (Macintosh; U; Intel Mac OS X**

- バイト数までが common format
- 第8 と第9フィールドが lighttpd の拡張
  - apache の combined format と共通

httpd ログ *(cont.)*

手順

1) 空白を中心にフィールドを取り出す
2) フィールド内最初がブランケットやダブルクォートなら対応する文字まで取り出す
   - 行が長いので、BUFSIZ をそのまま使うのではなく、XBUFSIZ とした
   - 必要に応じて BUFSIZ や任意の長さに変更する

libspji の rec_split 相当関数を用意する

```
#ifndef XBUFSIZ
#define XBUFSIZ (BUFSIZ)
#endif
int rec_split_httpdlog(rec_t *rc, char *src) {
  char *p, *q;
  int   c, fno;
  char  tmp[XBUFSIZ];
  int   sep=' ', isep=-1;

  p = src;
  fno = 0;
  while(*p && c<XBUFSIZ) {
```

```
    q = tmp;  c = 0;
    if(*p=='"'||*p=='[') {
      if(*p=='"') { isep = *p; }
      if(*p=='[') { isep = ']'; }
      p++;
      while(*p && c<XBUFSIZ && *p!=(char)isep) {
        *q++ = *p++; c++;  }
      if(*p!=(char)isep) {
        fprintf(stderr,
          "inside sepator not found\n");
      }
      p++;
    }
```

```
    else {
      while(*p && c<XBUFSIZ && *p!=(char)sep) {
        *q++ = *p++; c++;  }
    }
    *q = '\0';

    rec_setfield(rc, fno, tmp);
    fno++;
    if(*p && *p==(char)sep) {
      p++;  }
  }
  return fno;
}
```

読み込んだ行とレコード内容を表示してみる

```
% head -1 access.log | ./a.out
#123.45.67.89 localhost - [11/May/2008:08:10:33
 +0900] "GET /webplayer/index.py HTTP/1.1" 200
14637 "-" "Mozilla/5.0 (Windows; U; Windows NT
5.1; ja; rv:1.8.1.14) Gecko/20080404 Firefox/2.
0.0.14"
{"123.45.67.89","localhost","-","11/May/2008:08
:10:33 +0900","GET /webplayer/index.py HTTP/1.1
","200","14637","-","Mozilla/5.0 (Windows; U; W
indows NT 5.1; ja; rv:1.8.1.14) Gecko/20080404
Firefox/2.0.0.14","",}
```

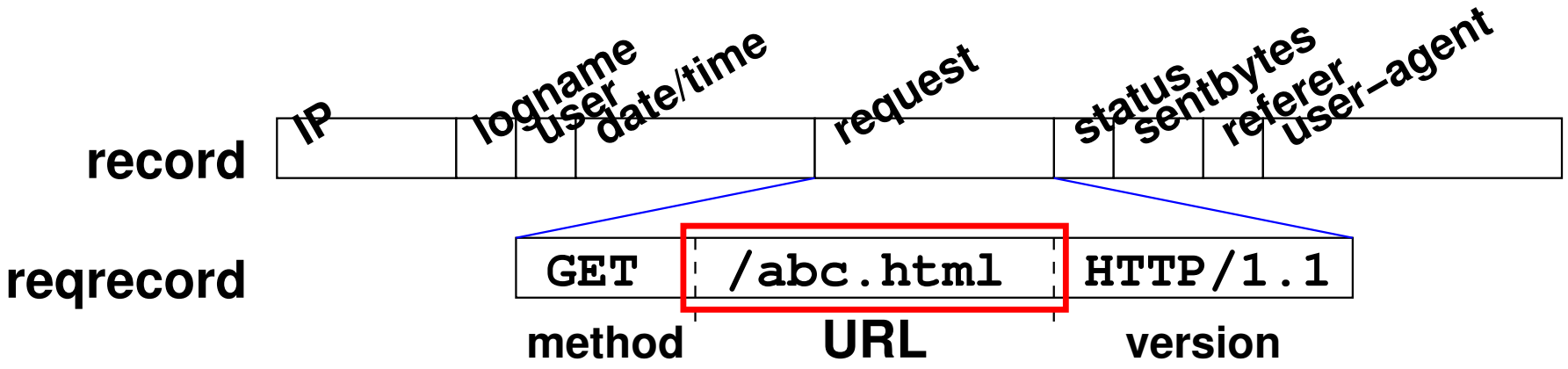
※ XBUFSIZ=4096; 最後のフィールドは行区切り

典型的なログ解析

- 頻繁にアクセスされている資源を見付ける
  - 資源毎の頻度分布
- アクセスの集中する時間帯を見付ける
  - 時間毎の頻度分布
  - 時間毎の転送バイト総和
- 頻繁にアクセスするクライアントを見付ける
  - クライアント毎の頻度分布

資源毎の頻度分布

- 資源を示す URL の頻度を計上
  - URL に関する辞書(idict) を設ける
- ログからリクエストを抜き出す
- リクエストから URL を抜き出す
  - ログやリクエストを分割するレコードを設ける



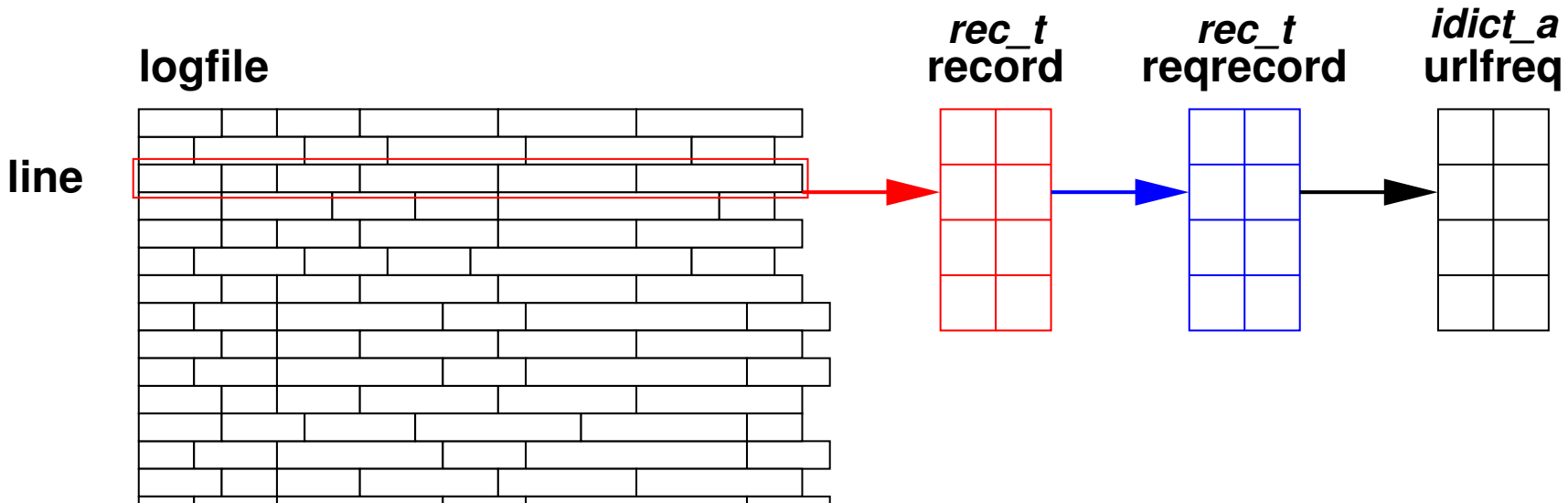
資源毎の頻度分布 *(cont.)*

辞書 urlfreq と頻度を計上する関数 stat_urlfreq

```
idict_a *urlfreq=NULL;
int stat_urlfreq(char *url) {
    idict_c *ipos;
    ipos = idict_findpos(urlfreq, url);
    if(ipos) { ipos->value++;  }
    else {     idict_add(urlfreq, url, 1);  }
}
```

```
int loop() {
  int lno;
  char line[XBUFSIZ];
  rec_t record, reqrecord;

  urlfreq = idict_new();
  idict_init(urlfreq);
  rec_init(&record);
  rec_clear(&record);
  rec_init(&reqrecord);
```



```
  lno = 0;
  while(fgets(line, XBUFSIZ, stdin)) {
    chomp(line);
    rec_split_httpdlog(&record, line);
    if(!record.fields[4]) {
      goto next;
    }
    rec_clear(&reqrecord);
    rec_split(&reqrecord, record.fields[4], ' ');
    if(!reqrecord.fields[1]) {
      goto next;
    }
    stat_urlfreq(reqrecord.fields[1]);
next:
```

```
    lno++;
  }
  qsort(urlfreq->slot, urlfreq->use,
    sizeof(urlfreq->slot[0]), idict_valuecmp);
  print_urlfreq();
}
```

最後の qsort で頻度順(昇順)に整列

前述の様に長い行がある、XBUFSIZ は SunOS 5.9 の BUFSIZ(1024)では足りない

```
%cc -DNDEBUG -DXBUFSIZ=4096 foo.c
```

資源毎の頻度分布 *(cont.)*

```
  0      1 /webplayer/contents/c2e/c2e1b687
...(略)
355     64 /favicon.ico
356     69 /webplayer/index.py
357     94 /webplayer/playlist.py
358   2300 /webplayer/comment.py
```

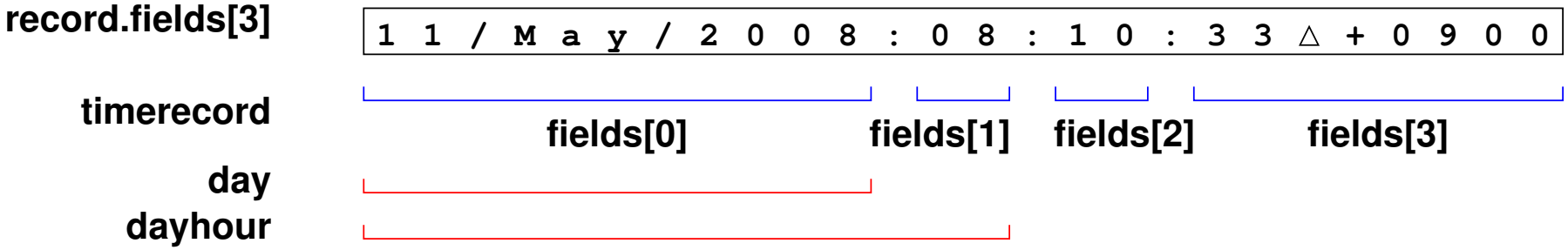
クライアント毎の頻度分布も同様

- record.field[0] を集計すれば良い
- reqrecord の処理は不要

時間毎の頻度分布

- 時間をキーに辞書で集計すれば良い
- 転送バイト総数もほぼ同様
  - バイト数分、カウンタをインクリメント

```
idict_a *timefreq;  rec_t timerecord;
int stat_timefreq(char *tstr);
...
  timefreq = idict_new();
  idict_init(timefreq);
  rec_init(&timerecord);
  ...
```




```
char day[BUFSIZ];
...
if(!record.fields[3]) {  goto next;  }
rec_clear(&timerecord);
rec_split(&timerecord, record.fields[3], ':');
day[0] = '\0';
if(!timerecord.fields[0]) { goto next; }
strcpy(day, timerecord.fields[0]);
stat_timefreq(day);
...
```

時間毎の頻度分布 *(cont.)*

- stat_timefreq() は stat_urlfreq() と同じ処理

実行結果

```
  0     2 15/Jun/2008
  1   247 13/Jun/2008
  2   456 12/Jun/2008
  3   668 09/Jun/2008
  4   727 10/Jun/2008
  5   805 14/Jun/2008
  6  1024 11/Jun/2008
  7  1236 08/Jun/2008
```

時間毎の頻度分布 *(cont.)*

キーに使う timerecord のフィールドを増やすと粒度が細かくなる

```
char dayhour[XBUFSIZ];
    ...
    if(!timerecord.fields[1]) { goto next; }
    strcpy(dayhour, day);
    strcat(dayhour, ":");
    strcat(dayhour, timerecord.fields[1]);
    ...
    stat_timefreq(dayhour);
```

時間毎の頻度分布 *(cont.)*

実行結果

```
  0      1 15/Jun/2008:04
  1      1 15/Jun/2008:00
...
 71    208 08/Jun/2008:23
 72    271 14/Jun/2008:18
 73    320 11/Jun/2008:17
```

分や秒を追加するとより細かくなる
時間帯の場合は timerecode.field[0] は加えず、field[1] や field[2] でキーを作ると良い

時間毎の転送バイト総数

バイト分追加する関数を設けて stat_timefreq の代わりに呼び出す

```
int stat_timeval(char *tstr, int val) {
    idict_c *ipos;
    ipos = idict_findpos(timefreq, tstr);
    if(ipos) {  ipos->value += val;    }
    else {      idict_add(timefreq, tstr, val); }
}
...
    stat_timeval(dayhour, atoi(record.fields[6]));
```

時間毎の転送バイト総数 *(cont.)*

実行結果(日・時間毎)

```
  0     31 10/Jun/2008:03
  1     31 13/Jun/2008:08
...
 71 543893041 13/Jun/2008:07
 72 897907318 09/Jun/2008:01
 73 1208431419 09/Jun/2008:00
```

※ 最後は 1.13ギガバイト

日毎

```
% ./a.out < access.log
  0    694 15/Jun/2008
  1 340993410 14/Jun/2008
  2 433866734 10/Jun/2008
  3 611269640 13/Jun/2008
  4 647566598 12/Jun/2008
  5 808820434 08/Jun/2008
  6 1161871437 11/Jun/2008
  7 -1532379821 09/Jun/2008
```

負の値が出て、結果が怪しい

long long

- 数百メガ、数ギガバイトのファイル転送が起きると...
- 総和はギガ以上、テラに達することもある
- int (典型的には 32 ビット) では集計できない
  - 符号なし 4 ギガ、符号付き ± 2 ギガ
- より大きな型が必要となる
  - long long (典型的には 64 ビット)
  - 符号なし 16 エクサ、符号付き ± 8 エクサ

→ long long を利用した辞書を作ると良い

idict から lldict を作る

- 機械的に value に関する型を int から long long にする

```
typedef struct {
    char *key;
    long long  value;
} lldict_c;
```

```
typedef struct {
    lldict_c *slot;
    int      len;
    int      use;
} lldict_a;
```

```
int lldict_add(lldict_a *dict, char *xkey,
        long long xvalue);
```

idict から lldict を作る *(cont.)*

- printf 族の整数表示は %d ではなく %lld を使う

```
printf("%3d %lld %s\n", i, timefreq->slot[i].valu
```

- 文字列から整数への変換は atoi や atol ではなく atoll を使う

```
stat_timeval(day, atoll(record.fields[6]));
```

- それ以外は idict そのまま

実行結果

```
% ./a.out < access.log
   0 694 15/Jun/2008
   1 340993410 14/Jun/2008
   2 433866734 10/Jun/2008
   3 611269640 13/Jun/2008
   4 647566598 12/Jun/2008
   5 808820434 08/Jun/2008
   6 1161871437 11/Jun/2008
   7 2762587475 09/Jun/2008
```

最後は 2.57ギガバイト

※ 先の結果は最後だけ間違っていた

大きな数字の視覚化

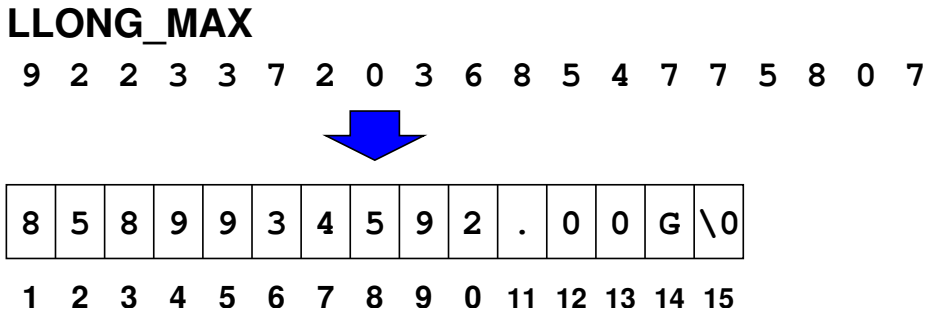
- 大きな数は直観的に把握できない
- 人間が読みやすいよう視覚化する
  - K, M, G をつける
  - 小数点以下を二桁つける

```
132432 -> 129.33K
1232423 -> 1.18M
12390239293 -> 11.54G
```

バッファオーバラン防止のため必要な文字列長を調査

```
% grep MAX /usr/include/*.h | grep LLONG
/usr/include/limits.h:#define   LLONG_MAX       9223372036854775807LL
/usr/include/limits.h:#define   ULLONG_MAX      18446744073709551615ULL
/usr/include/mlib_types.h:#define       MLIB_S64_MAX    LLONG_MAX
```

- 最大値の定義は 21文字、 ”LL”を抜いて19文字
- ギガは 10文字
- 最大値をギガバイト表記にすると 15文字必要
  - 数字で 11文字
  - 小数点以下で 3文字
  - K/M/G で 1文字
  - 終端文字で 1文字



```
int conv_ll2h(char*dst, int dlen, long long src) {
  if(dlen<15) {  return -1;  }
  if(src<1024) {
    sprintf(dst, "%lld", src);
  } else if(src<1024*1024) {
    sprintf(dst, "%.2fK", (double)src/(1024));
  } else if(src<1024*1024*1024) {
    sprintf(dst, "%.2fM",
      (double)src/(1024*1024));
  } else {
    sprintf(dst, "%.2fG",
      (double)src/(1024*1024*1024));
  }
  return 0;  }
```

実行結果（即値のif）

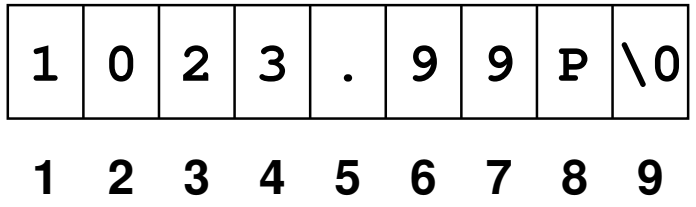
```
% ./a.out
37 -> 37
4122 -> 4.03K
1232423 -> 1.18M
2762587475 -> 2.57G
730429023929 3 -> 6802.65G
9334330420239293 -> 8693272.64G
59334330420239293 -> 55259401.37G
9223372036854775807 -> 8589934592.00G
```

成功
大きな数字がまだ対応できていない

配列を利用した実装に
即値の if は厄介、コード量と間違える可能性が増える

```
struct { char letter; long long base; } pf[] = {
  {'K', 1024LL}, {'M', 1024LL*1024},
  {'G', 1024LL*1024*1024},
  {'T', 1024LL*1024*1024*1024},
  {'P', 1024LL*1024*1024*1024*1024},
  {'E', 1024LL*1024*1024*1024*1024*1024},
#if 0
  {'Z', 1024LL*1024*1024*1024*1024*1024*1024},
  {'Y', 1024LL*1024*1024*1024*1024*1024*1024*1024},
#endif
};
```

```
int conv_ll2h(char*dst,int dlen,long long src) {
  int i, p = -1;
  if(dlen<9) { return -1; }
  for(i=0;i<sizeof(pf)/sizeof(pf[0]);i++) {
    if(src>pf[i].base) { p = i; }
  }
  if(p<0) { sprintf(dst, "%lld", src); }
  else {
    sprintf(dst, "%.2f%c",
      (double)src/pf[p].base, pf[p].letter);
  }
  return 0;
}
```



実行結果（配列で探索）

```
% ./a.out
37 -> 37
4122 -> 4.03K
1232423 -> 1.18M
2762587475 -> 2.57G
730429023929 3 -> 6.64T
9334330420239293 -> 8.29P
59334330420239293 -> 52.70P
9223372036854775807 -> 8.00E
```

大きな数字も対応できた

※Z や Y は long long の容量より大きいので保留

転送バイト総数の結果を読み易く

```
int print_timefreq() {
  int i, ck;
  char hmsg[BUFSIZ];
  for(i=0;i<timefreq->use;i++) {
    ck = conv_ll2h(hmsg, BUFSIZ,
           timefreq->slot[i].value);
    printf("%3d %lld %s %s\n", i,
      timefreq->slot[i].value, hmsg,
      timefreq->slot[i].key);
  }
}
```

転送バイト総数の結果を読み易く *(cont.)*

結果

```
% ./a.out < access.log
   0 694 694 15/Jun/2008
   1 340993410 325.20M 14/Jun/2008
   2 433866734 413.77M 10/Jun/2008
   3 611269640 582.95M 13/Jun/2008
   4 647566598 617.57M 12/Jun/2008
   5 808807202 771.34M 08/Jun/2008
   6 1161871437 1.08G 11/Jun/2008
   7 2762587475 2.57G 09/Jun/2008
```

演習

実際に解析してみよ
以下の場所にログファイルのサンプルがある
http://www2.jaist.ac.jp/is/private/lecture/i117/2008/access.log

1) アクセスの多い資源を降順に表示するプログラムを作れ★
2) クライアント毎のアクセス頻度を昇順に表示するプログラムを作れ★
3) 分毎の転送バイト総数を算出するプログラムを作れ★★

演習 *(cont.)*

4) 秒毎の転送バイト総数を算出するプログラムを作れ★★
5) 曜日毎の転送バイト総数を算出するプログラムを作れ★★★
   - データが8日分以上あれば、曜日について複数の結果がでるだろう、特に平均などの処理をする必要はない

演習 *(cont.)*

人間が読みやすい形式に変換

6) 実数の大きな値を K, M, G 等に変換する関数を作れ★

7) 実数の小さな値を m, u, n 等に変換する関数を作れ★★

演習 *(cont.)*

数式評価の応用

8) httpd のログを数式で各種加工するプログラムを作れ★★

9) 先のプログラムを帯域（単位時間辺りの転送バイト総数）を式で算出できるよう変更せよ★★★